改訂版

独習C++

ハーバート・シルト 著

CD-ROM付

SE
SHOEISHA

Teach Yourself C++ Third Edition

# 独習C++ 改訂版

ハーバート・シルト 著

トップスタジオ 訳

神林 靖 監修

CD-ROM付

# 独習C++

SE
SHOEISHA

## 本書の学習メソッド

**徹底的に演習を行うことにより,
プログラミング言語C++の基礎が
確実に身につきます.**

### 各章の構成

#### ❶ 前章の理解度チェック

新しい章の内容に入る前に,前の章で学んだ内容の理解度を確認するための問題を解きます.

**各節の構成**

新しい章のテーマをいくつかのトピックに分解しています.

❶ **解説**

❷ **例**

❸ **練習問題**

#### ❷ この章の理解度チェック

章末でその章全体の内容の理解度を確認するための問題を解きます.

#### ❸ 総合理解度チェック

新しい章で学んだ内容を,前の章までに学んだ知識に統合できたかを確認するための問題を解きます.

Teach Yourself C++ Third Edition

# 独習C++ 改訂版

ハーバート・シルト 著

トップスタジオ 訳

神林 靖 監修

独習C++

SE
SHOEISHA

## 本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

**●ご質問される前に**
弊社Webサイトの「Q&Aコーナー」(http://www.shoeisha.com/info/help.asp) をご参照ください。これまで受けたご質問への回答(FAQ) や、的確なご質問方法に関する情報を掲示しています。

**●ご質問方法**
弊社Webサイトの専用フォームサイト(http://www.shoeisha.com/book/qa/)をご利用ください。記載漏れや独自の用紙等によるご質問、お電話や電子メールによるお問い合わせ、本書にはさみ込まれたアンケートはがきに記入されたご質問等は、お受けしておりません。

**※質問専用シートのお取り寄せについて**
Webサイトにアクセスする手段をお持ちでない方は、ご氏名、ご送付先(ご住所/郵便番号/電話番号またはFAX番号/電子メールアドレス)および「質問専用シート送付希望」と明記のうえ、電子メール(qaform@shoeisha.com)、FAX、郵便(80円切手をご同封願います)のいずれかにて“編集部読者サポート係”までお申し込みください。お申し込まれた手段によって、折り返し質問シートをお送りいたします。シートに必要事項を漏れなく記入し、“編集部読者サポート係”までFAXまたは郵便にてご返送ください。

**●ご回答について**
ご回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

**●ご質問に際してのご注意**
本書の対象を越えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、予めご了承ください。

**●郵便物送付先およびFAX番号**

| | |
|---|---|
| 送付先住所 | 〒160-0006 東京都新宿区舟町5 |
| FAX番号 | 03-5362-3818 |
| 宛先 | (株)翔泳社出版局 編集部読者サポート係 |

・本書に記載されたURL等は予告なく変更される場合があります。
・本書の出版にあたっては正確な記述につとめましたが、著者や出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。
・本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

**Teach Yourself C++, Third Edition**
by Herbert Schildt

# はじめに

本書は，C 言語の知識をすでに持ち，さらにC++ を学ぼうとしている方を対象としています．

C++ は，オブジェクト指向プログラミング(OOP) を目指すCプログラマのための言語です．C という強固な基礎の上に構築されていて，Cの強力さ，優美さ，柔軟性はそのままに，OOPのサポート(並びにその他の新機能のサポート)を追加しています．C++は，すでに世界中のプログラマが使う普遍的プログラミング言語であり，次世代の高性能ソフトウェアは，この言語によって書かれることになるでしょう．職業的プログラマにとって最も重要な言語を1つ挙げるとすれば，それはC++ をおいてほかにありません．

C++ は，1979 年，米国ニュージャージー州マレーヒルのベル研究所で，Bjarne Stroustrup によって開発されました．最初は「C with classes (クラスを取り入れたC)」と呼ばれていましたが，1983 年にC++ と名称が変わりました．誕生以後，C++ は3 回の大きな改訂を経ています．最初が1985 年，2 回目が1990 年で，3 回目はC++ の標準化を通じての改訂です．1989 年にC++ の標準化作業が始まり，ANSI (米国規格協会) とISO (国際標準化機構) が共同でC++ 標準化委員会を発足させました．1994 年の1月25日には，各方面からの提案に基づき，第一草案が作成されました．ANSI/ISO C++委員会(筆者もメンバーの一人)が作成したこの草案では，Stroustrupが最初に定義した機能をそのまま保持しながら，新しい機能をいくつか追加しました．しかしこの第一草案は，おおむね当時のC++ の水準を反映したものにとどまっていました．

標準草案の完成後まもなく，この標準の大規模な拡張を促す出来事が起こりました．Alexander Stepanovが標準テンプレートライブラリ(STL)を作成したのです．本書で学ぶとおり，STLとは，データ操作に使用できる汎用ルーチンの集まりです．強力で，優美でしたが，巨大でもありました．第一草案の作成後，委員会は，C++ の仕様にSTL を含めることを決定しました．STL の追加によって，C++ の規模は最初の定義を大幅に超えるものとなりました．STL の取り込みは重要なことではありましたが，反面，C++ 標準化の遅れという副作用ももたらしました．

C++の標準化にこれほど長い時間がかかるとは，着手当時，誰も予想していなかったでしょう．ようやく，1997 年末に最終草案が委員会を通過し，正式承認を待つのみです．実質的に，標準C++ はすでに現実のものになっています．多くのコンパイラは，新しい機能をすべてサポートするように変わりつつあります．

本書は，標準C++を解説しています．標準C++は，ANSI/ISO標準化委員会によって作成されたC++であり，現在，すべての主要コンパイラで受け入れられているC++ です．本書を手に取るプログラマは，今日学んだことを，自信をもって明日使うことができるのです．

## 第3版で新しくなったこと

本書は，*Teach yourself C++*（『独習C++』）の第3版（日本語版は改訂版）です．前2版の内容をすべて含み，2つの新しい章と，多数の新トピックを付け加えています．新しく設けられた章の1つでは，実行時型情報（RTTI）と新しいキャスト演算子を説明しています．もう1つの章では，標準テンプレートライブラリ（STL）を説明しています．どちらも，第2版の発行後にC++言語に追加された重要機能です．新しいトピックとしては，名前空間，新型ヘッダー，新しいスタイルの入出力システムがあります．このため第3版は，以前の2版より相当分量が多くなりました．

## Windowsを使用している方に

Windowsベースのマシンを使用し，Windowsベースのプログラムを書こうとしている方にとって，C++を学ぶことは正しい選択です．C++は，Windowsプログラミングによくなじみます．ただし，本書に示すプログラム例はすべてコンソールベースのプログラムであり，Windowsプログラムは含まれていません．その理由は単純明快で，Windowsプログラムは，その性質上，大きくて複雑になりがちだからです．骨格だけのWindowsプログラムを作成するのに，50～70行が必要です．C++の機能を実際に試すWindowsプログラムを書くには，数百行ものコードが必要になります．率直に言って，プログラミングの学習環境としてWindowsは適切とは言えません．しかし本書のプログラムは，Windowsベースのコンパイラでコンパイルできます．

C++をマスターすれば，Windowsプログラミングにもその知識を応用できます．実際，C++を使ったWindowsプログラミングでは，MFCなどのクラスライブラリを使用できるため，Windowsプログラムの開発を非常に容易に行うことができます．

## 本書の構成

本書のユニークな点は，技能上達学習理論（Mastery Learning Theory）を適用してC++を学習していくことです．一度に1つの概念を提示し，いくつもの例で確認し，実際に練習問題を解くことによりマスターします．この方法では，1つのトピックを完全に理解してから次のトピックに移ることができます．

内容は順次に提示され，1つ1つ積み重ねられていきます．したがって，どの章も注意深く読み進んでください．どの章でも，それ以前のすべての章の内容がすでに身についていることを想定しています．第1章を除く各章では，冒頭の「前章の理解度チェック」で，前章で得た知識を確認します．各章の末尾には「この章の理解度チェック」があり，その章の内容についての知識を確認します．最後に，「総合理解度チェック」で，新しく学んだ内容とそれ以前の章で学んだ知識を統合できたかを確認します．

本書では，熟練したCプログラマを読者に想定しています．率直に言って，Cプログラミングができなければ，C++プログラミングもできません．Cでプログラムを書くことができない方は，本書をお使いになる前に，まずCを学習してください．Cを学習するには，筆者の『独習C』(小社刊)をお勧めします．本書と同じ学習メソッドを採用しています．

## 謝辞


本書の作成中に知識とアドバイス，専門的な意見を与えてくれた次の方々に感謝します．

**Bjarne Stroustrup** **Steve Clamage**

**P.J.Plauger** **Al Stevens**

# 付属 CD-ROM の使い方

## ■ 付属 CD-ROM の内容

本書の付属 CD-ROM には，本書で掲載しているサンプルプログラムを章別のテキストファイルに収録しています．たとえば，第 12 章にあるサンプルプログラムは「ch21.txt」という名前のファイルに，第 12 章にある問題の解答は「ans12.txt」に含まれています．

なお，win フォルダには SJIS コードのファイル，unix フォルダには EUC コードのファイルが，それぞれ格納されています．内容はまったく同じです．

## ■ 動作環境

CD-ROM が対応しているオペレーティングシステムは，UNIX，Windows 95，Windows 98，Windows NT 3.51 以上です．

本書のサンプルプログラムをコンパイルし，実行するには，C++ コンパイラが必要です．コンパイル，ビルド，実行などの方法の詳細は，処理系によって異なりますので，お使いのコンパイラのユーザーズマニュアルなどを参照してください．

なお，Microsoft 社の Visual C++ 6.0，および INPRISE 社の Borland C++Builder 4 以降では，標準入出力においてかな漢字がサポートされておりますので，cout，cin でかな漢字を使用することができます．

## ■ Visual C++ を使ってサンプルプログラムを実行する方法

コンパイラとして Visual C++ を使用する場合は，次のようにしてください．

1. コンソールアプリケーション用のプロジェクトを生成する
2. 作成したプロジェクトに，空の拡張子 .cpp のソースコードモジュールを追加する
3. テキストエディタを使って，章別のサンプルテキストからプログラムコードをコピーして，作成した空のソースコードモジュールに貼り付ける
4. ビルドする
5. 実行する

# 目 次

# 1

# C++ の概要

## この章の内容

C++ 言語はC言語の拡張版です．C++にはCのすべての機能が含まれており，それに加えてオブジェクト指向プログラミングもサポートされています．オブジェクト指向プログラミングという点を除いても，C++には単純に「より良いC言語」と呼ぶことができるような多くの改善点と機能が含まれています．ごく一部の例外を除いて，C++はCのスーパーセットです．Cに関する知識はすべてC++にも同様に当てはまりますが，強化されたC++の機能を理解するためには，相当の時間と努力を費やさなければなりません．しかし，C++でプログラミングを行うことによって，費やした努力を上回る見返りが得られることは間違いありません．

この章の目的は，C++の中で特に重要ないくつかの機能を紹介することです．ご存じのとおり，コンピュータ言語を構成する個々の要素というものは，ばらばらに存在するわけではありません．これらの要素すべてが集まって1つの言語を形成します．C++ではこの相互関係が特に顕著です．C++の各機能は密接に統合されているので，C++の1つの面について単独で説明するのは困難です．そこで，この章ではC++のいくつかの機能について概要を示します．全体を大まかに理解しておけば，本書で後述する例を理解するのに役立つでしょう．ほとんどの主題については，後続の章でさらに詳しく説明します．

C++はオブジェクト指向プログラミングをサポートする目的で開発されたため，この章ではまずオブジェクト指向プログラミングについて説明します．C++の大部分の機能は，何らかの形でオブジェクト指向プログラミングに関係しています．オブジェクト指向プログラミングの理論はC++に浸透しているのです．ただし，C++を使って，オブジェクト指向ではないプログラムを作成することもできることを覚えておいてください．C++をどのように使うかは，完全にプログラマ次第なのです．

本書の執筆時点では，C++の標準化が完成しつつあります．そこで，この章では，ここ数年の間に広く使われてきたバージョンのC++と，新しい標準C++との主な相違について説明します．本書は標準C++の入門書なので，古いコンパイラを使用する読者にとってはこれらの事項が特に重要となるでしょう．

C++の重要な機能の紹介に加え，この章ではCとC++のプログラミングスタイルの相違についても解説します．C++では，さまざまな面で，より柔軟な方法でプログラムを作成することができます．これらの機能の中には，オブジェクト指向プログラミングとはほとんど(またはまったく)関係がないものもありますが，大部分のC++プログラムで使われる機能です．したがって，本書の最初の方で説明することにします．

本題に入る前に，C++の性質と形式について一般的なことを述べておきます．まず，C++プログラムの外見は多くの点でCプログラムと似ています．Cプログラムと同様，C++プログラムの実行はmain()から始まります．コマンドライン引数を含めるためには，Cと同じargc，argvを使用します．C++では独自のオブジェクト指向ライブラリが定義されていますが，C標準ライブラリのすべての関数もサポートされています．C++ではCと同じ制御構造を使用します．C++には，Cで定義されている組み込みのデータ型がすべて含まれています．

本書では，読者がCプログラミング言語について知っているものと想定しています．つまり，本書を使用してC++によるプログラミングを学ぶには，Cによるプログラミングの知識を持っていなければなりません．Cの知識がない方は，筆者の著書である『Teach Yourself C, Third Edition』(Osborne/McGraw-Hill, 1997；邦訳：『独習C 改訂版』，小社刊)を読んで学習することをお勧めします．同書では，本書と同じ学習理論を採用しており，C言語全体について詳しく解説しています．

メモ

本書では，読者がC++コンパイラを使用してプログラムをコンパイルして実行する方法を知っているものと想定しています．これらの方法を知らない場合は，使用するコンパイラのマニュアルを参照してください(コンパイルの方法はコンパイラによって異なるので，本書ですべて説明することはできません)．プログラミングを学習する最良の方法は，実際に試してみることです．本書で紹介する例を順次入力し，コンパイルし，実行することをお勧めします．

## 1.1 オブジェクト指向プログラミングとは

**オブジェクト指向プログラミング**(Object-Oriented Programming: OOP)とは，プログラミングに取り組むための強力な手法です．プログラミングには，初期の頃からさまざまな方法論が使われてきました．プログラミングの発展における節目では，ますます複雑になるプログラムに対処するために，新しいアプローチが開発されてきました．最初のプログラムは，コンピュータのフロントパネルにあるスイッチをオンとオフの間で切り替えることによって作成されました．しかし，この手法はごく短いプログラムにしか適さないことは明らかです．次にアセンブリ言語が開発され，これによってより長いプログラムを作成できるようになりました．次の進歩がもたらされたのは，1950年代，最初の高レベル言語(FORTRAN)が開発されたときでした．

高レベル言語を使うことによって，プログラマは数千行にわたるプログラムを作成することができるようになりました．しかし，初期に使われていたプログラミング方式はアドホックであり，何でも許される方式でした．比較的小さなプログラムならばこれでもかまいませんが，大きなプログラムにこの方法を使用すると，読みづらい(そして保守しにくい)「スパゲティコード」が出来上がりました．1960年代には，**構造化プログラミング言語**(structured programming language)の登場によって，スパゲティコードを排除することが可能になりました．この種の言語にはAlgolやPascalがあります．広い意味ではCも構造化言語であり，これまでに読者が使用してきたプログラミング言語は大部分が構造化プログラミングと呼ばれる類のものでしょう．構造化プログラミングでは，厳密に定義された制御構造，コードブロックを使用し，GOTOを使わず(あるいは，少なくともその使用を最小限に抑え)，再帰とローカル変数をサポートする独立したサブルーチンを使用します．構造化プログラミングの要点

は，プログラムをその構成要素に簡約することです．構造化プログラミングの手法を用いれば，汎庸なプログラマでも50,000行に及ぶ長さのプログラムを作成し，保守することができます．

構造化プログラミングは，適度な複雑さのプログラムに適用したときは優れた効果をもたらしますが，プログラムのサイズがある程度に達すると限界が現れます．より複雑なプログラムを作成するためには，新しいプログラミング手法が必要となります．そこで開発されたのがオブジェクト指向プログラミングです．オブジェクト指向プログラミングでは，構造化プログラミングに見られる優れた概念を取り入れ，それを，プログラムをより効率的に編成するための強力な新しい概念と組み合わせました．オブジェクト指向プログラミングでは，問題点をさらに細かい構成要素に分解することができます．それぞれの構成要素は自己完結したオブジェクトであり，そのオブジェクトに関する独自の指示とデータを持っています．これによって複雑さが軽減され，プログラマはより大きなプログラムを扱えるようになります．

C++を含め，すべてのオブジェクト指向プログラミング言語は，カプセル化，ポリモーフィズム，継承という3つの特色を共通して備えています．次に，これらの概念について説明しましょう．

## カプセル化

**カプセル化**(encapsulation) は，プログラムコードとプログラムコードが扱うデータを一体化して，外部の干渉や誤用から両者を保護するためのしくみです．オブジェクト指向言語では，プログラムコードとデータを組み合わせて，自己完結型の「ブラックボックス」を作成することができます．このようにしてプログラムコードとデータを組み合わせることによって，**オブジェクト**(object)が作成されます．つまり，オブジェクトとはカプセル化をサポートするための工夫なのです．

オブジェクト内では，プログラムコードとデータ，またはその両方が，そのオブジェクトに対して非公開か公開のどちらかになります．**非公開**(private)のプログラムコードとデータはオブジェクト内のほかの部分から認識し，アクセスすることができます．つまり，非公開のプログラムコードとデータには，オブジェクトの外部にあるプログラム内からアクセスすることができません．これに対して，**公開**(public)のプログラムコードとデータは，オブジェクト内で定義されているにもかかわらず，ほかのプログラム内からアクセスすることができます．一般に，オブジェクトの公開部分は，オブジェクトの非公開な部分への制御インターフェイスを提供するために使われます．

どのような使い方をする場合でも，オブジェクトはユーザー定義型の変数です．プログラムコードとデータを結び付けるオブジェクトを変数と考えるのは，奇妙に感じられるかもしれません．しかし，オブジェクト指向プログラミングではまさにこのとおりなのです．新しい種類のオブジェクトを定義するたびに，新しいデータ型を作成していることになります．このデータ型の各インスタンスは複合変数です．

## ポリモーフィズム

**ポリモーフィズム**(polymorphism. ギリシア語で「多数の形態」の意)とは，1つの名前を2つまたはそれ以上の関連する(技術的には異なる)目的に使用できるようにする性質のことです．オブジェクト指向プログラミングでは，ポリモーフィズムを使うことによって，1つの名前で複数の動作の汎用クラスを指定することができます．実際にどの動作を適用するかは，データの型によって決まります．たとえば，ポリモーフィズムを明確にサポートしていないC言語の場合，絶対値を求めるためにはabs()，labs()，fabs()という3つの別々の関数名を使わなければなりません．これらの関数は，それぞれ整数，長整数，浮動少数点数値の絶対値を計算して返します．しかし，ポリモーフィズムをサポートしているC++の場合は，これらの各関数を1つの名前(abs()など)で呼び出すことができます(実際にこれを行う方法については，この章の後半で説明します)．実際にどの関数を実行するかは，関数を呼び出す際に使用したデータの型によって決まります．このように，C++では1つの関数名を複数の異なる目的に使用することができるのです．このことを**関数のオーバーロード**(function overloading)と呼びます．

一般的な言い方をすると，ポリモーフィズムは「1つのインターフェイスで複数のメソッドを使用する」という概念によって表すことができます．つまり，関連する一連の動作に対して，汎用インターフェイスを使用するということです．ポリモーフィズムの利点は，1つのインターフェイスで複数の動作の汎用クラスを指定することにより，複雑さを減らすことができるという点です．それぞれの状況でどの動作を適用するかを選択するのは，コンパイラの役目です．プログラマがこの選択を手作業で行う必要はありません．プログラマは汎用インターフェイスがあることを覚えておき，それを使用するだけでいいのです．前述の例に示したように，絶対値を求める関数に，1つで済むはずが3つもの名前があると，数値の絶対値を求めるという当たり前の動作が実際よりも複雑になってしまいます．

ポリモーフィズムは演算子にも当てはめることができます．ほとんどすべてのプログラミング言語では，数値演算にポリモーフィズムをある程度適用しています．たとえば，Cの+記号は整数，長整数，文字，浮動少数点数値の加算に使われます．この場合，実際にどの演算を実行するかはコンパイラが判断します．この種のポリモーフィズムは，**演算子のオーバーロード**(operator overloading)と呼ばれます．

ポリモーフィズムについて覚えておくべき重要な点は，ポリモーフィズムを使うと，関連する複数の機能に対して標準インターフェイスを作成できるため，より複雑なプログラムに対処できるということです．

## 継承

**継承**(inheritance)とは，1つのオブジェクトがほかのオブジェクトの特色を獲得するプロセスのことです．具体的に言うと，オブジェクトは特定の性質を汎用セットとして受け継いだ上で，そのオブ

ジェクト独自の機能を追加することができるということです．継承という概念を使用すると，**階層的な分類**(hierarchical classification) という概念をオブジェクトがサポートできるようになります．大部分の情報は，階層的な分類によって管理できます．たとえば，「家」の説明について考えてみてください．家は「建物」という汎用クラスの一部です．建物はさらに汎用的な「建築物」クラスの一部です．そして，建築物はさらに汎用的な「人工物」クラスの一部です．どのクラスでも，子クラスは親クラスが持つすべての性質を継承し，さらに独自の性質を追加して定義しています．階層的な分類を使用しなければ，各オブジェクトに関連する性質をすべて明示的に定義しなければなりません．これに対して，継承を使用すると，そのオブジェクトが所属する汎用クラス(複数の場合もある)と，そのオブジェクト独自の性質を指定することによって，1つのオブジェクトを記述することができます．このように，継承はオブジェクト指向プログラミングの中で非常に重要な役割を果たしています．

## 例　1.1　オブジェクト指向プログラミングとは

1. カプセル化は，オブジェクト指向プログラミングにおいてまったく新しい概念というわけではありません．Cを使用しても，ある程度のカプセル化は行うことができます．たとえばライブラリ関数を使うときには，実際にはブラックボックスルーチンを使っているわけで，不正な行為を行わない限りは，その内部を修正したり影響を与えたりすることはできません．fopen()関数について考えてみましょう．この関数を使用してファイルを開く際には，いくつかの内部変数が作成され，初期化されます．プログラム側から見る限りでは，これらの変数は隠されており，アクセスすることができません．しかしC++では，より安全なカプセル化の方法が用意されています．

2. 実世界ではポリモーフィズムの例が非常によく見られます．たとえば，自動車のハンドルについて考えてみましょう．パワーステアリングであっても，ラックアンドピニオン式であっても，マニュアル式であっても，同じ働きをします．重要なのは，実際のどのステアリング機構(方式)を使用する場合でも，インターフェイス(ハンドル)は同じだということです．

3. 性質の継承とより汎用的な分類の概念は，知識を組織化する際の基本的な方法です．たとえば，セロリは野菜クラスのメンバであり，野菜クラスは植物クラスのメンバです．さらに，植物クラスは生物クラスのメンバです．階層的な分類がなければ，知識の組織化は不可能です．

**練習問題　1.1　オブジェクト指向プログラミングとは**

**1.** 分類とポリモーフィズムが，私たちの日常生活の中で果たしている重要な役割について考えなさい．

## 1.2　2つのバージョンのC++

本書の序文でも説明したとおり，過去数年の間，C++の標準化が進められてきました．その目的は，次世代のプログラミングニーズに適合した，安定して標準化された機能豊富な言語を作成することです．その結果，C++には実際には2つのバージョンが存在します．その1つは，Bjarne Stroustrupの当初の設計を基盤とした従来のバージョンです．これは過去10年にわたって多くのプログラマに使われてきたバージョンのC++です．もう1つは，新しい標準C++です．これはStroustrupとANSI/ISO標準化委員会によって作成されました．これらの2つのバージョンのC++は，中枢部分は非常によく似ていますが，標準C++には従来のC++に見られない拡張機能がいくつか追加されています．したがって標準C++は，本質的に従来のC++のスーパーセットであると言えます．

本書では標準C++について説明します．これはANSI/ISO標準化委員会によって定義されたバージョンのC++であり，最近のすべてのC++コンパイラでサポートされています．本書で紹介するプログラムコードは，標準C++によって奨励されている現在のコーディングスタイルとコーディング習慣を反映しています．つまり，本書で学ぶ内容は，現在だけでなく将来的にも通用するということです．率直に言って，標準C++は将来の言語です．そして，標準C++は以前のバージョンのC++が備えていたすべての機能を包括しているので，本書で学んだC++は，すべてのC++プログラミング環境で役に立ちます．

ただし，古いコンパイラを使用している場合は，本書で紹介するすべてのプログラムをコンパイルすることはできません．これは，標準化の過程で，ANSI/ISO委員会がこの言語に新しい機能を数多く追加したからです．これらの機能は，追加されるにつれて，コンパイラ開発者によって実装されていきました．当然ながら，言語に新しい機能を追加してから，市販のコンパイラでその機能を使用できるようになるまでには，必ず時間のずれがあります．新機能は数年の期間にわたってC++に追加されたので，それらのすべてはサポートしていない古いコンパイラもあります．C++に最近追加された2つの新機能は，作成するすべてのプログラム（ごく単純なプログラムであっても）に関係するものなので，このことが重要になります．ただし，これらの新機能をサポートしていない古いコンパイラを使用している場合でも，簡単な対処法があるので心配はいりません．詳細については，以降の節で説明していきます．

古いスタイルと新しいスタイルの相違として，2つの新機能に関するものがあります．2つの新機能とは，新しいスタイルヘッダとnamespace文です．これらの相違を例示するために，ここでは何も行

わない最小限のC++プログラムを2種類示します．次に示す1つ目のプログラムでは，つい最近まで使われていた方法(古いコーディングスタイル)でC++プログラムを記述しています．

```
/*
    従来のC++プログラムのコメントスタイル
*/
#include <iostream.h>

int main()
{
  /* プログラムコード */
  return 0;
}
```

C++はCを基盤として作成されているので，このスケルトンの大部分についてはお馴染みでしょう．このプログラムの#include文に注目してください．この文ではiostream.hというファイルをインクルードしています．これはC++の入出力システムをサポートするためのファイルで，Cではstdio.hに当たります．

次に，新しいスタイルを使用したスケルトンプログラムを示します．

```
/*
    新しいスタイルのヘッダと名前空間を使用する
    新しいスタイルのC++プログラム
*/
#include <iostream>
using namespace std;

int main()
{
  /* プログラムコード */
  return 0;
}
```

このプログラムの最初のコメントの直後にある2行に注目してください．この部分が，1つ目のプログラムと異なっています．まず，#include文にはiostreamという名前の後に.hがありません．さらに，次の名前空間を指定する行が新しく挿入されています．これらの新しいスタイルヘッダについては本書で詳しく説明しますが，ここでも簡単に説明しておくことにしましょう．

## 新しいC++ヘッダ

C言語によるプログラミングを経験している読者は，プログラムでライブラリ関数を使うときには，そのヘッダファイルをインクルード(include)しなければならないということはご存じでしょう．この

ためには#include 文を使用します．たとえば，C で入出力関数用のヘッダファイルをインクルードするには，次の文を使用して stdio.h ファイルをインクルードします．

```
#include <stdio.h>
```

stdio.hは，入出力関数で使用するファイルの名前です．このファイル名の前の部分にある文によって，このファイルがプログラムにインクルードされます．ここで重要なのは，#include文によってファイルがインクルードされるということです．

C++が最初に開発されてから数年の間は，Cと同じスタイルのヘッダが使われていました．実際に，標準C++でも，プログラマが作成したヘッダファイルをサポートするため，そして下方互換性を維持するために，Cスタイルのヘッダは依然としてサポートされています．しかし，標準C++では標準C++ライブラリが使用する新しい種類のヘッダが追加されました．新しいスタイルのヘッダでは，ファイル名を指定せずに，標準識別子のみを指定します．標準識別子はコンパイラによってファイルにマッピングされますが，これは必須ではありません．新しいスタイルのC++ヘッダは，C++ライブラリで必要とされる適切なプロトタイプと定義を確実に宣言するための抽象概念です．

新しいスタイルのヘッダはファイル名ではないので，.h拡張子がありません．新しいヘッダは，ヘッダ名だけが山括弧(<>)に囲まれた形をしています．次に，標準C++でサポートされている新しいスタイルのヘッダの例を示します．

**新しいスタイルのヘッダ**

```
<iostream>
<fstream>
<vector>
<string>
```

新しいスタイルのヘッダをインクルードするには，#include 文を使用します．唯一の相違点は，新しいスタイルのヘッダでは必ずしもファイル名を指定する必要がないということです．

C++ にはC 関数ライブラリ全体が含まれているので，これらのライブラリに関連付けられた標準Cスタイルのヘッダファイルも，依然としてサポートされています．つまり，stdio.h や ctype.h といったヘッダもまだ使用できます．それに加えて，標準C++では新しいヘッダスタイルも定義されており，ヘッダファイルの代わりに使用することもできます．C++バージョンの標準Cヘッダでは，ファイル名の前にcというプレフィックスが付き，拡張子.hは付きません．たとえば，math.hは新しいスタイルのC++ ヘッダでは<cmath>，string.h は<cstring> となります．現在のところは，C ライブラリを使用するときには，Cスタイルのヘッダファイルをインクルードすることも許可されていますが，これは標準C++では推奨されていません．このため，本書ではすべての#include文で，新しいスタイルのC++ヘッダを使用します．C関数ライブラリ用の新しいヘッダスタイルがサポートされていないコンパイラを使う場合は，古いC のヘッダに置き換えてください．

新しいヘッダスタイルはC++にごく最近追加された機能なので，このスタイルを使用していない古いプログラムは数多く存在します．これらのプログラムではCスタイルのヘッダを使用し，ファイル名を指定します．前述の古いスタイルのスケルトンプログラムからもわかるように，従来の方法では次のようにして入出力ヘッダをインクルードします．

```
#include <iostream.h>
```

これによってiostream.hファイルがプログラムにインクルードされます．一般に，古いスタイルのヘッダでは，新しいスタイルの対応するヘッダと同じ名前を使用し，それに.hを追加します．

本書の執筆時点では，すべてのC++コンパイラは古いヘッダスタイルをサポートしています．しかし，古いヘッダスタイルは「旧式」と公言されており，新しいプログラムで使用することは推奨されていません．したがって，本書では古いヘッダスタイルを使用しません．

**重要** 古いヘッダスタイルは，まだ多くのC++プログラムで使われていますが，「旧式」のスタイルとして公言されています．

## 名前空間

新しいスタイルのヘッダをプログラムにインクルードすると，そのヘッダの内容がstd名前空間に含まれます．**名前空間**(namespace)とは，単に宣言された領域のことです．名前空間の目的は，名前の競合を防ぐために，識別子の名前を局所化することです．これまでは，ライブラリ関数の名前およびこの種の項目は，単純にグローバルな名前空間に置かれていました(Cと同様)．しかし，新しいスタイルのヘッダの内容はstd名前空間に置かれます．名前空間については，本書の後半で詳しく説明します．ここでは，次の文を使用してstd名前空間を可視状態にする(stdをグローバルな名前空間に入れる)ことができるので，名前空間について理解していなくてもかまいません．

```
using namespace std;
```

この文がコンパイルされた後は，古いヘッダスタイルと新しいヘッダスタイルの相違はなくなります．

## 古いコンパイラの使用

前述のとおり，名前空間と新しいヘッダスタイルは，どちらもC++に最近追加された機能です．新しいC++コンパイラならば，ほとんどすべてがこれらの機能をサポートしていますが，古いコンパイラではサポートされていない可能性があります．古いコンパイラを使用している場合は，本書のサンプルプログラムをコンパイルする際に，最初の2行で1つまたは複数のエラーが発生します．その場合は，

単に古いヘッダスタイルに置き換え，namespace文を削除してください．つまり，次のような2行を，

```
#include <iostream>
using namespace std;
```

次の１行に差し替えます．

```
#include <iostream.h>
```

この修正によって，新しいプログラムが従来のスタイルのプログラムに変わります．古いヘッダスタイルでは，そのすべての内容をグローバル名前空間に読み込むので，namespace文は必要ありません．

これから数年の間は，古いヘッダスタイルを使用し，namespace文を含んでいないC++プログラムも数多く残ることでしょう．C++コンパイラでは，これらのプログラムも問題なくコンパイルできます．しかし，標準C++では新しいスタイルしかコンパイルできないので，新しく作成するプログラムでは新しいスタイルを使用してください．しばらくは古いスタイルのプログラムもサポートされますが，技術の流れに逆行する方法と言えます．

**練習問題　1.2　2つのバージョンのC++**

1. 次に進む前に，前述の新しいスタイルのスケルトンをコンパイルしてみてください．このプログラムでは何の処理も実行しませんが，コンパイルすると，使用しているコンパイラでC++の新しい構文がサポートされているかどうかがわかります．新しいヘッダスタイルやnamespace文が受け入れられなかった場合は，前述の方法に従って，古いヘッダスタイルに置き換えてください．新しいスタイルをサポートしていないコンパイラを使う場合は，本書のすべてのプログラムでこの修正作業を忘れずに行ってください．

## 1.3　C++のコンソール入出力

C++はCのスーパーセットなので，Cのすべての要素はC++にも含まれています．つまり，すべてのCプログラムは，デフォルトではC++プログラムでもあります(実際には，この規則に当てはまらないごく少数の例外もあります．この点については本書の後の章で説明します)．したがって，Cプログラムとまったく同じに見えるC++プログラムを作成することもできます．このこと自体は誤りではありませんが，それではC++の利点を完全には活かすことができません．C++を最大限に活用するには，

C++スタイルのプログラムを作成しなければなりません．つまり，C++独自のコーディングスタイルと機能を使用するということです．

多くのC++プログラマが使用する，最も一般的なC++独自の機能は，おそらくコンソール入出力でしょう．printf()やscanf()といった関数を使うこともできますが，C++では新しく改善された方法で入出力操作を行うことができます．C++では，入出力関数の代わりに入出力演算子を使用して入出力操作を行います．出力演算子は<<，入力演算子は>>です．ご存じのとおり，これらはCではそれぞれ左と右のシフト演算子として使われています．C++でも，これらの演算子には左右のシフト演算子としての役割も残っていますが，機能が拡張されて，入出力も実行するようになっているのです．次のC++の文について考えてみましょう．

```
cout << "This string is output to the screen.¥n";
```

この文では，文字列をコンピュータの画面に表示しています．coutは既定義のストリームで，C++プログラムの実行が始まったときに，自動的にコンソールにリンクされます．これはCのstdoutと似ています．Cと同様，C++のコンソール入出力もリダイレクトすることができますが，ここではコンソールを使っているものと想定して説明を進めます．

<<出力演算子を使うと，C++の任意の基本的なデータ型を出力することができます．たとえば次の文では100.99という値を出力しています．

```
cout << 100.99;
```

一般に，コンソールに出力する際には，次の形式で<<演算子を使用します．

**<<演算子**

```
cout << expression;
```

*expression*には，C++の任意の有効な式を指定します．ほかの出力式を指定することもできます．

キーボードから入力を受け取るには，>>入力演算子を使用します．たとえば，次のプログラムコードでは，整数値をnumに格納しています．

```
int num;
cin >> num;
```

numの前に&が付いていない点に注目してください．Cでscanf()関数を使用して値を入力するときには，変数のアドレスを関数に渡すことによって，ユーザーからの入力値を受け取りました．C++の入力演算子を使う場合は，これとは異なります(その理由については，C++について学んでいくうちに明らかになります)．

一般に，キーボードから値を入力するには，次の形式で>>演算子を使用します．

>> 演算子

```
cin >> variable;
```

メモ

拡張された<<演算子と>>演算子は，演算子オーバーロードの例でもあります．C++の入出力演算子を使うには，プログラムに<iostream>ヘッダをインクルードする必要があります．前述のとおり，これはC++の標準ヘッダの 1 つであり，C++ コンパイラによって提供されます．

## 例 1.3　C++ のコンソール入出力

**1.** 次のプログラムは，1つの文字列，2つの整数値，1つの倍精度浮動少数点数値を出力します．

```
#include <iostream>
using namespace std;

int main()
{
  int i, j;
  double d;

  i = 10;
  j = 20;
  d = 99.101;

  cout << "Here are some values: ";
  cout << i;
  cout << ' ';
  cout << j;
  cout << ' ';
  cout << d;

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Here are some values: 10 20 99.101
```

重要

古いコンパイラを使用している場合は，このプログラムおよび本書で紹介するほかのプログラムで使っている新しいヘッダスタイルとnamespace文がサポートされていない可能性があります．これらがサポートされていない場合は，前の節で説明した古いスタイルのプログラムコードに置き換えてください．

**2.** 1つの入出力式で複数の値を出力することができます．次のプログラムは，例1で示したプログラムを修正したもので，入出力文の効率的な記述方法を示しています．

```
#include <iostream>
using namespace std;

int main()
{
  int i, j;
  double d;

  i = 10;
  j = 20;
  d = 99.101;

  cout << "Here are some values: ";
  cout << i << ' ' << j << ' ' << d;

  return 0;
}
```

このプログラムの次の行では，1つの式で複数の項目を出力しています．

```
cout << i << ' '<< j << ' ' << d;
```

一般には，1つの文を使用して，項目を好きなだけいくつでも出力することができます．これがわかりにくい場合は，<< 出力演算子はC++のほかの演算子と同じように動作し，任意の長さの式の中で使用できると考えてください．

項目と項目の間に空白を出力したいときは，明示的に空白を指定する必要があります．空白を指定しないと，各項目は連続して画面上に表示されます．

**3.** 次のプログラムでは，ユーザーに整数値の入力を求めます．

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  cout << "Enter a value: ";
  cin >> i;
  cout << "Here's your number: " << i << "¥n";

  return 0;
}
```

このプログラムからの出力例を次に示します．

```
Enter a value: 100
Here's your number: 100
```

ご覧のとおり，ユーザーが入力した値が変数 i に格納されます．

**4.** 次のプログラムでは，ユーザーに整数値，浮動少数点数値，文字列の入力を求めます．このプログラムでは，1つの入力文を使用して3つすべての値を読み取ります．

```
#include <iostream>
using namespace std;

int main()
{
  int i;
  float f;
  char s[80];

  cout << "Enter an integer, float, and string: ";
  cin >> i >> f >> s;
  cout << "Here's your data: ";
  cout << i << ' ' << f << ' ' << s;

  return 0;
}
```

この例からもわかるように，1つの入力文で，必要なだけいくつでも項目を入力することができます．C と同様，各データ項目は空白文字(スペース，タブ，改行)によって区切る必要があります．

文字列を読み取る場合，1つ目の空白文字が見つかると入力が止まります．たとえば，このプログラムに対して次のように入力したとします．

```
10 100.12 This is a test
```

この場合，プログラムからの出力は次のようになります．

```
10 100.12 This
```

文字列が不完全なのは，文字列の読み取りが This の後で止まっているからです．文字列の残りの部分は，入力バッファに残され，次の入力操作を待機します(これは，%s 形式の scanf() を使用して文字列を入力する場合と似ています)．

**5.** デフォルトでは，>>を使用すると，すべての入力では行バッファが使われます．つまり，ユーザーが［Enter］キーを押すまで，情報はC++プログラムに渡されません(Cのscanf()でも行バッファが使われるので，この入力方式は目新しいものではないでしょう)．行バッファ入力の影響を確認するために，次のプログラムを実行してみてください．

```
#include <iostream>
using namespace std;

int main()
{
  char ch;

  cout << "Enter keys, x to stop.¥n";
  do {
    cout << ": ";
    cin >> ch;
  } while (ch != 'x');

  return 0;
}
```

このプログラムを実行し，入力した文字をプログラムに送信するためには，文字キーを押した後に毎回［Enter］キーを押す必要があります．

## 練習問題　1.3　C++のコンソール入出力

**1.** 従業員の労働時間とその賃金を入力し，その従業員の合計賃金を表示するプログラムを作成しなさい(入力プロンプトを表示すること)．

**2.** フィートをインチに変換するプログラムを作成しなさい．ユーザーにフィート数の入力を求め，それに対応するインチ数を表示します．ユーザーがフィート数として0を入力するまで，この処理を繰り返します．

**3.** 次にCプログラムを示します．C++スタイルの入出力文を使用して，このプログラムを書き直しなさい．

```
/* このCプログラムをC++スタイルに書き直す
   このプログラムは，最小公分母を計算する
*/
#include <stdio.h>
```

```
int main(void)
{
  int a, b, d, min;

  printf("Enter two numbers: ");
  scanf("%d%d", &a, &b);
  min = a > b ? b : a;
  for(d = 2; d<min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
   printf("No common denominators¥n");
    return 0;
  }
  printf("The lowest common denominator is %d¥n", d);

  return 0;
}
```

# 1.4　C++のコメント

C++では，プログラムに2とおりの方法でコメントを含めることができます．1つ目の方法では，標準的なC形式のコメントを使用します．つまり，コメントの始めに/*を記述し，最後に*/を記述します．Cと同様，C++でもこの種のコメントをネスト(入れ子)にすることはできません．

2つ目の方法では，単一行コメントを使用します．単一行コメントは//で始まり，その行の最後で終了します．単一行コメントでは，物理的な行末(キャリッジリターン/ラインフィードの組み合わせ)以外には，コメント終了記号は使用しません．

一般にC++プログラマは，複数行にわたるコメントについてはC形式のコメントを使用し，短いコメントについては単一行コメントを使用します．

## 例　1.4　C++のコメント

**1.** 次のプログラムには，CスタイルのコメントとC++スタイルのコメントの両方を使用しています．

```
/*
    これはC形式のコメント
    このプログラムは，整数が奇数か偶数かを判別する
*/
#include <iostream>
```

```
using namespace std;

int main()
{
  int num;  // これはC++形式の単一行コメント

  // 数値を読み取る
  cout << "Enter number to be tested: ";
  cin >> num;

  // 奇数か偶数かを調べる
  if((num%2)==0) cout << "Number is even¥n";
  else cout << "Number is odd¥n";

  return 0;
}
```

**2.** 複数行コメントをネストにすることはできませんが，複数行コメント内で単一行コメントをネストにすることはできます．たとえば，次のコメントは完全に有効です．

```
/* これは複数行コメントである
   このコメント内で，//単一行コメントをネストにすることができる
   複数行コメントはここで終わる
*/
```

複数行コメント内で単一行コメントをネストにすることができるので，デバッグ用に数行のプログラムコードをコメント化しやすくなります．

**練習問題　1.4　C++のコメント**

**1.** 次のコメントが有効かどうかを考えなさい(C++形式の単一行コメント内で，C形式のコメントをネストにしています)．

```
// 変わったコメントの /* 記述方法 */
```

**2.** 1.3節の練習問題の解答に，自分でコメントを追加しなさい．

# 1.5　クラス

C++の重要な機能を1つだけ挙げるとすれば，おそらく**クラス**(class) でしょう．クラスとは，オブジェクトを作成するために使われるしくみのことです．したがって，クラスはC++の数多くの機能の

核心に当たります．クラスについてはほかの章でも詳しく説明しますが，C++プログラミングにおいてクラスは基本となるので，ここでも簡単に触れておきます．

クラスを宣言するにはclassキーワードを使用します．クラスの宣言構文は，構造体の宣言構文に似ています．クラス宣言の一般形式を次に示します．

**クラス宣言**

```
class class-name {
    // 非公開関数と変数
public:
    // 公開関数と変数
}  object-list;
```

このクラス宣言の*object-list*は任意指定です．構造体の場合と同様に，クラスオブジェクトは後から必要に応じて宣言することができます．技術的には*class-name*の部分も任意指定ですが，事実上はほとんど必須です．その理由は，クラス名はそのクラスのオブジェクトを宣言するために使われる新しい型名となるからです．

クラス宣言内で宣言される関数と変数は，そのクラスの**メンバ**(member)と呼ばれます．デフォルトでは，クラス宣言内で宣言されたすべての関数と変数はそのクラスに対して非公開です．つまりこれらの関数と変数には，そのクラスのほかのメンバからしかアクセスすることができません．公開クラスメンバを宣言するには，public キーワードを指定し，その後にコロンを追加します．public キーワードを指定した後に宣言したすべての関数と変数には，そのクラスのほかのメンバからでも，そのクラスを含むプログラムのほかの部分からでもアクセスすることができます．

次に，単純なクラス宣言の例を示します．

```
class myclass {
  // myclassクラス内で非公開
  int a;
public:
  void set_a(int num);
  int get_a();
};
```

このクラスには，a という 1 つの非公開変数と，set_a()および get_a()という 2 つの公開関数があります．これらの関数はクラス内でプロトタイプ形式を使って宣言されています．クラスの一部として宣言された関数を**メンバ関数**(member function) と呼びます．

aは非公開変数なので，myclassクラスの外部にあるプログラムコードからはアクセスすることができません．set_a()関数と get_a()関数は myclass クラスのメンバなので，変数 a にアクセスすること

ができます．さらに，get_a()関数とset_a()関数はmyclassクラスの公開メンバとして宣言されているので，myclassクラスを含むプログラムのほかの部分からでも呼び出すことができます．

get_a()関数とset_a()関数は，myclassクラスで宣言されていますが，定義はされていません．メンバ関数を定義するには，そのクラスの型名を関数名にリンクする必要があります．このためには，関数名の前にクラス名と2つのコロンを付けます．この2つのコロン(::)は**スコープ解決演算子**(scope resolution operator)と呼ばれます．次に，set_a()関数とget_a()関数の定義例を示します．

```
void myclass::set_a(int num)
{
  a = num;
}

int myclass::get_a()
{
  return a;
}
```

set_a()関数とget_a()関数はどちらも，myclassクラスの非公開変数である変数aにアクセスすることができます．set_a()関数とget_a()関数はmyclassクラスのメンバなので，非公開データに直接アクセスすることができます．

一般に，メンバ関数を定義するには次の一般形式を使用します．

**メンバ関数**

```
ret-type class-name::func-name(parameter-list)
{
   // 関数の本体
}
```

*class-name*の部分にはその関数が所属するクラスの名前を指定します．

myclassクラスの宣言では，myclass型のオブジェクトについては何も定義していません．実際に宣言したときに作成されるオブジェクトの型を定義しているだけです．オブジェクトを作成するにはクラス名を型指定子として使用します．次に示す行では，myclass型の2つのオブジェクトを宣言しています．

```
myclass ob1, ob2; // これらはmyclass型のオブジェクトである
```

**重要** クラス宣言は，新しい型を定義する論理的な抽象作用です．クラス宣言では，その型のオブジェクトの外観を定義します．オブジェクト宣言では，その型の物理エンティティを作成します．つまり，オブジェクトはメモリ領域を占有しますが，型定義はメモリを占有しません．

あるクラスのオブジェクトを作成すると，構造体メンバにアクセスする場合と同じように，ドット(.)演算子を使用してその公開メンバにアクセスすることができます．前述のオブジェクト宣言の場合，次の文を使用して，ob1オブジェクトとob2オブジェクトのset_a()メソッドを呼び出すことができます．

```
ob1.set_a(10); // ob1オブジェクトの変数aを10に設定する
ob2.set_a(99); // ob2オブジェクトの変数aを99に設定する
```

コメントにも示したとおり，これらの文はob1オブジェクトの変数aを10に，ob2オブジェクトの変数aを99に設定しています．各オブジェクトは，クラス内で宣言されたすべてのデータの独自のコピーを持っています．つまり，ob1オブジェクトの変数aは，ob2オブジェクトの変数aとは別のものなのです．

**重要** クラスの各オブジェクトは，そのクラス内で宣言されたすべての変数の独自のコピーを持っています．

## 例 1.5 クラス

**1.** 次のプログラムでは，本文中に示したmyclassクラスを使用し，ob1オブジェクトとob2オブジェクトの変数aの値を設定して，各オブジェクトの変数aの値を表示しています．

```
#include <iostream>
using namespace std;

class myclass {
  // myclassクラス内で非公開
  int a;
public:
  void set_a(int num);
  int get_a();
};

void myclass::set_a(int num)
{
  a = num;
}

int myclass::get_a()
{
  return a;
}
```

```
int main()
{
  myclass ob1, ob2;

  ob1.set_a(10);
  ob2.set_a(99);

  cout << ob1.get_a() << "¥n";
  cout << ob2.get_a() << "¥n";

  return 0;
}
```

このプログラムを実行すると，10と99が画面上に表示されます．

**2.** 例1のmyclassクラスの変数aは非公開変数です．つまり，変数aには，myclassクラスのメンバ関数からしか直接アクセスすることができません(そのために，公開関数であるget_a()関数が必要となります)．あるクラスの非公開メンバに，そのクラスのメンバ以外の部分からアクセスしようとすると，コンパイル時にエラーが発生します．例1のmyclassクラスが定義されている場合，次のmain()関数を使用するとエラーが発生します．

```
// このプログラムコードにはエラーが含まれている
#include <iostream>
using namespace std;

int main()
{
  myclass ob1, ob2;

  ob1.a = 10; // エラー．メンバ関数以外からは
  ob2.a = 99; // 非公開メンバにアクセスできない

  cout << ob1.get_a() << "¥n";
  cout << ob2.get_a() << "¥n";

  return 0;
}
```

**3.** 公開メンバ関数を作成できるのと同じように，公開メンバ変数を作成することもできます．次に示すように，変数aをmyclassクラスのpublicセクションで宣言すると，プログラム内のどこからでも変数aを参照することができます．

```
#include <iostream>
using namespace std;
```

```
class myclass {
public:
  // aは公開変数
  int a;
  // したがって，set_a()関数とget_a()関数は必要ない
};

int main()
{
  myclass ob1, ob2;

  // 変数aは直接アクセスされる
  ob1.a = 10;
  ob2.a = 99;

  cout << ob1.a << "¥n";
  cout << ob2.a << "¥n";

  return 0;
}
```

この例では変数aをmyclassクラスの公開メンバとして宣言しているので，main()関数から直接アクセスすることができます．変数aにアクセスする際のドット演算子の使い方に注目してください．一般に，クラスの外部からメンバ関数を呼び出すとき，またはメンバ変数にアクセスするときには，オブジェクト名の後にドット演算子を追加し，その後にメンバの名前を指定して，どのオブジェクトメンバを参照するかを明確に指定する必要があります．

**4.** オブジェクトの有効性を実感できるように，より実用的な例を紹介します．次のプログラムでは，stackというクラスを作成し，このクラスを文字を保存するスタックとして使います．

```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字を保存するstackクラスを宣言する
class stack {
  char stck[SIZE]; // スタック領域を確保する
  int tos; // スタック先頭の索引
public:
  void init(); // スタックを初期化する
  void push(char ch); // スタックに文字をプッシュする
  char pop(); // スタックから文字をポップする
```

```
};

// スタックを初期化する
void stack::init()
{
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty";
    return 0; // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack s1, s2;  // 2つのスタックを作成する
  int i;

  // スタックを初期化する
  s1.init();
  s2.init();
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "¥n";

  return 0;
}
```

このプログラムからの出力を次に示します.

```
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
```

次に，このプログラムの内容を詳しく説明することにしましょう．classクラスには，stckとtosの2つの非公開変数が含まれています．stck配列には，実際にはスタックにプッシュされた文字が保存されており，変数tosにはスタックの先頭の索引が保存されています．stackクラスには公開関数としてinit()，push()，pop()があります．これらはそれぞれ，スタックの初期化，値のプッシュ，値のポップを行います.

main()関数内では，s1とs2の2つのスタックを作成し，各スタックに3つの文字をプッシュします．各スタックオブジェクトは別のものであるということを確実に理解してください．つまり，s1に文字をプッシュしても，s2にプッシュした文字には何の影響もありません．各オブジェクトには，stckとtosの独自のコピーが含まれています．この概念は，オブジェクトを理解する上では基本となります．同じクラスのオブジェクトはすべて同じメンバ関数を持っていますが，各オブジェクトは独自のデータを作成し，管理します.

## 練習問題　1.5　クラス

**1.** この節の例をまた試していない場合は，ここで入力して実行しなさい.

**2.** 図書館の図書目録の項目を管理するための，cardという名前のクラスを作成しなさい．このクラスに本のタイトル，著者，在庫冊数を格納します．タイトルと著者を文字列として保存し，在庫冊数を整数として保存します．store()という名前の公開メンバ関数を使用して本の情報を保存し，show()という名前の公開メンバ関数を使用して情報を表示します．このクラスの動作を確認するために，単純なmain()関数を追加しなさい.

**3.** 整数の循環キューを管理するキューのクラスを作成しなさい．キューのサイズを整数100個の大きさにします．このクラスの動作を確認するために，単純なmain()関数を追加しなさい.

## 1.6　CとC++の相違点

C++はCのスーパーセットではありますが，両者の間には小さな相違点がいくつかあります．そのうちのいくつかは最初に覚えておくべきものです．先に進む前に，ここでそれらの相違点について説明しておきます．

まずCでは，関数で仮引数を受け取らないとき，プロトタイプの関数仮引数リストにはvoidというキーワードを記述します．たとえば，Cでf1()という関数が仮引数を受け取らず，char型のデータを返す場合，そのプロトタイプは次のようになります．

```
char f1(void);
```

これに対して，C++ではvoidキーワードの指定が任意です．したがって，C++ではf1()関数のプロトタイプを一般に次のように書きます．

```
char f1();
```

C++では，Cと違って空の仮引数リストを指定することができます．このプロトタイプをCで使った場合は，その関数の仮引数に関しては何も記述していないことになります．しかしC++では，関数に仮引数がないことを意味します．したがって，この例では空の仮引数リストを宣言するために，明示的にvoidを使用していません(voidを使用して空の仮引数リストを宣言するのは，規則に反するわけではありません．単に冗長なだけです．多くのC++プログラマは効率を追求するのに熱心なため，このようにvoidを使ったプログラムを見ることはほとんどありません)．C++では，次の2つの宣言は同じ意味になります．

```
char f1();
char f1(void);
```

CとC++の第2の相違点としては，C++では，すべての関数のプロトタイプを記述しなければならないという点があります．Cでは，プロトタイプは推奨されてはいましたが，技術的には任意指定です．C++ではプロトタイプは必須です．前節の例からもわかるように，クラスに含まれるメンバ関数のプロトタイプは，全般的なプロトタイプとして使われるので，このほかに別のプロトタイプを用意する必要はありません．

CとC++の第3の相違点としては，C++では，関数が戻り値を返すものとして宣言した場合，その関数は必ず戻り値を返さなければなりません．つまり，void以外の戻り値型を持つ関数では，関数内のreturn文に値を必ず含める必要があります．Cでは，void以外の値を返す関数で，実際に値を返さなくてもかまいません．実際に値を返さない場合は，内容の保証されない値が「返され」ます．

Cでは，関数の戻り値の型を明示的に指定しないと，整数型が使われます．C++ではデフォルト値を整数型とする規則がなくなりました．したがって，すべてのC++関数では戻り値の型を明示的に宣言しなければなりません．

このほかのCとC++の相違点として，ローカル変数を宣言できる場所に関するものがあります．Cでは，ローカル変数はブロックの先頭で，どの「アクション」文よりも前に宣言しなければなりません．C++では，ローカル変数はどこでも宣言できます．この方法の利点は，ローカル変数を最初に使用する場所の近くで宣言できるので，意図しなかった影響が及ぶのを防ぐことができるということです．

最後に，C++ではboolデータ型が定義されています．これはブール値(真/偽)を保存するために使われます．C++では，trueキーワード(真)とfalse(偽)キーワードも定義されており，bool型の値にはこれらの値しか格納することができません．C++では，関係演算子と論理演算子の結果はbool型の値であり，すべての条件文の評価結果はブール値にならなければなりません．Cと比べて，これは一見大きな違いのように思えるかもしれませんが，そうではありません．実際には，ほとんど意識せずに済みます．ご存じのとおり，Cの真は非ゼロであり，偽はゼロです．これはC++でも同じことで，C++ではブール式を評価する際に，すべての非ゼロは真に，すべてのゼロは偽に自動的に変換されます．逆方向の変換が行われることもあります．整数式でブール値を使用すると，真は1に，偽は0に変換されます．bool型が追加されたことによって，データ型チェックをより徹底的に行うことができ，ブール型と整数型を区別することができるようになりました．もちろん，実際に使用するかどうかは任意です．bool型は好都合なデータ型です．

## 例 1.6 CとC++の相違点

**1.** Cプログラムでは，コマンドライン引数を受け取らない場合，次のmain()関数を宣言するのが一般的です．

```
int main(void)
```

しかしC++では，voidキーワードの使用は冗長であり，不要です．

**2.** 次の短いC++プログラムは，sum()関数のプロトタイプを記述していないので，コンパイルすることができません．

```
// このプログラムはコンパイルされない
#include <iostream>
using namespace std;

int main()
{
```

```
  int a, b, c;

  cout << "Enter two numbers: ";
  cin >> a >> b;
  c = sum(a, b);
  cout << "Sum is: " << c;

  return 0;
}

// この関数にはプロトタイプが必要である
sum(int a, int b)
{
  return a+b;
}
```

**3.** 次の短いプログラムからわかるように，ローカル変数はブロック内のどこでも宣言することができます．

```
#include <iostream>
using namespace std;

int main()
{
  int i; // ローカル変数をブロックの先頭で宣言する

  cout << "Enter number: ";
  cin >> i;

  // 階乗を計算する
  int j, fact=1; // アクション文の後に変数を宣言する

  for(j=i; j>=1; j--) fact = fact * j;
  cout << "Factorial is " << fact;

  return 0;
}
```

変数jと変数factを，最初に使用する場所の近くで宣言することの利点は，この短いプログラムではあまり得られません．しかし大きな関数では，最初に使用する場所の近くで変数を宣言するとプログラムコードが明確になり，予期しない副作用を防ぐのに役立ちます．

**4.** 次のプログラムでは，outcomeという名前のbool型変数を作成し，false値を代入しています．次に，if文でこの変数を使用しています．

```
#include <iostream>
using namespace std;

int main()
{
  bool outcome;

  outcome = false;

  if(outcome) cout << "true";
  else cout << "false";

  return 0;
}
```

このプログラムによって，false が表示されます．

## 練習問題 1.6 CとC++の相違点

**1.** 次のプログラムをC++プログラムとしてコンパイルすることはできません．理由を説明しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
using namespace std;

int main()
{
  f();
  return 0;
}

void f()
{
  cout << "this won't work";
}
```

**2.** C++プログラム内のさまざまな場所で，ローカル変数の宣言を試しなさい．Cプログラムでも同じことを試し，どの場合にエラーが発生するかを確認しなさい．

# 1.7　関数のオーバーロード

C++の機能でクラスの次に重要な機能は，おそらく関数を多重定義すること，つまり**関数のオーバーロード**(function overloading) でしょう．関数のオーバーロードによって，C++である種のポリモーフィズムを実現できるばかりでなく，C++プログラミング環境を随時拡張していくための基盤も形成されます．オーバーロードは非常に重要な機能なので，ここで簡単に触れておきます．

C++では，2つまたはそれ以上の関数に同じ名前を使用することができます．ただし，それぞれの引数の型か引数の数(またはその両方)が異なっていなければなりません．2つ以上の関数で同じ名前を使用するとき，それらはオーバーロードされていると表現します．関数をオーバーロードすると，関連する複数の操作を同じ名前で参照できるので，プログラムの複雑さを減らすことができます．

関数をオーバーロードするのはごく簡単です．必要なそれぞれの関数を宣言し，定義するだけです．関数を呼び出す際に使われた引数の数またはデータ型を基に，コンパイラによって適切な関数が自動的に選択されます．

メモ　C++では，演算子をオーバーロードすることもできます．ただし，演算子のオーバーロードについて完全に理解するには，C++についてさらに学ぶ必要があります．

## 例　1.7　関数のオーバーロード

1. 関数オーバーロードの主な用途として，コンパイル時ポリモーフィズムを行うことが挙げられます．これによって，1つのインターフェイスで多くのメソッドを使用できるようになります．ご存じのとおり，Cプログラミングでは，処理するデータ型が違うだけで，似たような機能を持つ関数が数多くあります．この典型的な例が，C標準ライブラリに見られます．前述のとおり，ライブラリにはabs()，labs()，fabs()という3つの関数が含まれており，それぞれ整数，長整数，浮動少数点数の絶対値を返します．

   しかし，3種類のデータ型を処理するためには3種類の名前が必要となるので，状況は必要以上に複雑になります．どの関数も絶対値を返し，データ型が違うだけです．次の例に示すように，C++では，3種類のデータ型に対して1つの名前をオーバーロードすることによって，この状況に対処することができます．

```
#include <iostream>
using namespace std;
```

```
// abs()関数を3とおりにオーバーロードする
int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
  cout << "Absolute value of -10: " << abs(-10) << "¥n¥n";
  cout << "Absolute value of -10L: " << abs(-10L) << "¥n¥n";
  cout << "Absolute value of -10.01: " << abs(-10.01)
  ➥<< "¥n¥n";
  return 0;
}

// 整数用のabs()関数
int abs(int n)
{
  cout << "In integer abs()¥n";
  return n<0 ? -n : n;
}

// 長整数用のabs()関数
long abs(long n)
{
  cout << "In long abs()¥n";
  return n<0 ? -n : n;
}

// 倍精度浮動少数点数用のabs()関数
double abs(double n)
{
  cout << "In double abs()¥n";
  return n<0 ? -n : n;
}
```

このプログラムでは，abs()という名前で3つ(各データ型用に1つずつ)の関数を定義しています．main()関数では，3種類の引数を使用してabs()関数を呼び出しています．コンパイラは，引数として使われたデータ型を基に，適切なabs()関数を呼び出します．このプログラムからの出力を次に示します．

```
In integer abs()
Absolute value of -10: 10
In long abs()
Absolute value of -10L: 10
In double abs()
Absolute value of -10.01: 10.01
```

この例は単純なものですが，それでも関数オーバーロードの価値がよくわかります．1つの名前を使用してある動作の汎用クラスを記述することができるので，ほんの少し異なる3つの名前(abs()，fabs()，labs())を使用することによって発生する不自然な複雑さを除外することができます．プログラマは，一般的な動作を表す1つの名前だけを覚えればよいことになります．呼び出すべき適切な関数(メソッド)を選ぶのは，コンパイラの役目です．これによって，複雑さが減るという効果が得られます．つまり，この例では，ポリモーフィズムを使うことによって，3つの名前が1つに減りました．

この例では，ポリモーフィズムの使用効果は少ししかわかりませんが，大きなプログラムになると，「1つのインターフェイスで複数のメソッドを使用する」というアプローチが非常に効果的に実行されていることがよくわかります．

**2.** 次のプログラムは，関数オーバーロードのもう1つの例です．この例では，date()関数をオーバーロードし，日付を文字列としても，3つの整数としても受け取るようにしています．どちらの場合も，関数で受け取った日付を画面に表示します．

```
#include <iostream>
using namespace std;

void date(char *date); // 文字列の日付
void date(int month, int day, int year); // 数値の日付

int main()
{
  date("8/23/99");
  date(8, 23, 99);
  return 0;
}

// 文字列の日付
void date(char *date)
{
  cout << "Date: " << date << "¥n";
}

// 整数の日付
void date(int month, int day, int year)
{
  cout << "Date: " << month << "/";
  cout << day << "/" << year << "¥n";
}
```

この例では，関数のオーバーロードによって，関数に対する非常に自然なインターフェイスを作成しています．日付は，文字列として表現することも，月，日，年を含む3つの整数として表現することも，どちらもごく一般的なので，プログラマはそのときどきの状況によって便利な方の形式を自由に選択することができます．

**3.** これまでは，引数のデータ型が異なる関数のオーバーロードを見てきました．しかし，関数のオーバーロードには，次に示すように，引数の数が異なるものもあります．

```
#include <iostream>
using namespace std;

void f1(int a);
void f1(int a, int b);

int main()
{
  f1(10);
  f1(10, 20);
  return 0;
}

void f1(int a)
{
  cout << "In f1(int a)¥n";
}

void f1(int a, int b)
{
  cout << "In f1(int a, int b)¥n";
}
```

**4.** 関数のオーバーロードを行うためには，各関数の戻り値のデータ型が異なるだけでは不十分です．2つの関数の相違点が戻り値のデータ型だけだとすると，関数が呼び出されたときに，コンパイラは常に正しい関数を呼び出すことができないからです．次に示すコード例は，この点が不明瞭なので，正しくありません．

```
// これは正しくないのでコンパイルできない
int f1(int a);
double f1(int a);
.
.
.
f1(10); // コンパイラが呼び出すべき関数を判断できない
```

コメントに示したように，コンパイラはどちらのf1()関数を呼び出すべきかを判断できません．

## 練習問題　1.7　関数のオーバーロード

**1.** 引数の平方根を返す，sroot()という名前の関数を作成しなさい．sroot()関数を3とおりにオーバーロードし，それぞれ整数，長整数，倍精度浮動少数点数の平方根を返すようにしなさい(平方根を実際に計算するには，標準ライブラリ関数のsqrt()を使用します)．

**2.** C++標準ライブラリには，次の3つの関数が含まれています．

```
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
```

これらの関数は，sが指す文字列に含まれている数値を返します．atof()関数は倍精度浮動少数点数を，atoi()関数は整数を，atol()関数は長整数を返します．これらの関数はオーバーロードできません．その理由を説明しなさい．

**3.** 引数として受け取った2つの数値のうち，小さい方を返すmin()という名前の関数を作成しなさい．min()関数をオーバーロードし，それぞれ文字，整数，倍精度浮動少数点数を引数として受け取るようにしなさい．

**4.** 引数として受け取った秒数だけコンピュータを停止するsleep()という名前の関数を作成しなさい．sleep()関数をオーバーロードし，整数と，整数の文字列表現のどちらを使用しても呼び出せるようにしなさい．たとえば次の関数呼び出しでは，どちらもコンピュータを10秒間停止します．

```
sleep(10);
sleep("10");
```

これらの関数を小さなプログラム内に組み込み，動作を確認しなさい(コンピュータを停止させるために，遅延ループを使用してもかまいません)．

# 1.8　C++のキーワード

C++では，Cで定義されているすべてのキーワードをサポートしており，さらに独自のキーワードが30個追加されています．表1-1に，C++で定義されている全キーワードを示します．

C++の初期のバージョンではoverloadキーワードが定義されていましたが，これは旧式のキーワードです．

表 1-1　C++ のキーワード

| | | | | | |
|---|---|---|---|---|---|
| asm | const_cast | explicit | int | register | switch |
| union | auto | continue | extern | long | reinterpret_cast |
| template | unsigned | bool | default | false | mutable |
| return | this | using | break | delete | float |
| namespace | short | throw | virtual | case | do |
| for | new | signed | true | void | catch |
| double | friend | operator | sizeof | try | volatile |
| char | dynamic_cast | goto | private | static | typedef |
| wchar_t | class | else | if | protected | static_cast |
| typeid | while | const | enum | inline | public |
| struct | typename | | | | |

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. ポリモーフィズム，カプセル化，継承について簡単に説明しなさい．

2. C++ プログラムにコメントを含める方法を説明しなさい．

3. C++ 形式の入出力を使用して，キーボードから2つの整数を入力し，1つ目の整数に対する2つ目の整数乗を表示するプログラムを作成しなさい(たとえば，ユーザーが2と4を入力した場合は，$2^4$，つまり16を表示します)．

4. 文字列を逆順にする rev_str() という名前の関数を作成しなさい．rev_str() 関数をオーバーロードし，引数として文字列を1つでも2つでも渡すことができるようにしなさい．文字列を1つ指定して呼び出した場合は，その文字列に逆順の文字列を格納します．文字列を2つ指定して呼び出した場合は，2つ目の引数に逆順の文字列を返します．次に例を示します．

```
char s1[80], s2[80];
strcpy(s1, "hello");
rev_str(s1, s2); // s1は修正せずに，逆順にした文字列をs2に格納する
rev_str(s1);     // 逆順にした文字列をs1に返す
```

5. 次の新しいスタイルのC++プログラムを，古いスタイルのプログラムに変更する方法を説明しなさい．

```
#include <iostream>
using namespace std;

int f(int a);

int main()
{
  cout << f(10);
  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

**6.**　boolデータ型とは何か説明しなさい.

# 2

# クラスの概要

## この章の内容

この章では，クラスとオブジェクトについて説明します．C++プログラミングのほぼ全体に関係する重要なトピックもいくつか紹介するので，慎重に読むことをお勧めします．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. C++形式の入出力を使用し，ユーザーに文字列の入力を求め，その文字列の長さを表示するプログラムを作成しなさい．

2. 名前とアドレス情報を保存するクラスを作成しなさい．すべての情報をクラスの非公開メンバに文字列として保存します．名前とアドレスを保存する公開関数，名前とアドレスを表示する公開関数を作成します(これらの関数には，store()とdisplay()という名前を付けます)．

3. 引数として受け取ったビットを左に回転した上で，結果を返すオーバーロード関数rotate()を作成しなさい．整数と長整数を受け取るように，この関数をオーバーロードしなさい(回転はシフトと似ていますが，シフトにより一端から外れた文字をもう一端から入れる点が異なります)．

4. 次のプログラムコードの誤りを指摘しなさい．

```
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
.
.
.
};

int main()
{
  myclass ob;
  ob.i = 10;
.
```

```
.
.
}
```

## 2.1　コンストラクタ関数とデストラクタ関数

プログラミング経験の長い方であれば，プログラム中で初期化が必要な状況があることをご存じでしょう．オブジェクトを扱うときには，初期化の必要性はさらに高まります．実際，作成するほとんどすべてのオブジェクトには，何らかの初期化作業が必要です．そのためにC++では，クラス宣言に**コンストラクタ関数**(constructor function)を含めることができます．クラスのコンストラクタは，そのクラスのオブジェクトが作成されるたびに呼び出されます．したがって，そのオブジェクトに対して必要なすべての初期化処理を，コンストラクタ関数によって自動的に実行することができるのです．

コンストラクタ関数は，それが所属するクラスと同じ名前を持ち，戻り値型を持ちません．次に，コンストラクタ関数を含むクラスの簡単な例を示します．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(); // コンストラクタ
  void show();
};

myclass::myclass()
{
  cout << "In constructor¥n";
  a = 10;
}

void myclass::show()
{
  cout << a;
}

int main()
{
  myclass ob;

  ob.show();
```

```
  return 0;
}
```

この単純な例では，変数aの値をコンストラクタmyclass()内で初期化しています．コンストラクタはオブジェクトの作成時に呼び出されます．オブジェクトは，そのオブジェクトの宣言文が実行されたときに作成されます．C++では，変数宣言文は「実行文」であることを覚えておいてください．Cでプログラムを作成する際には，宣言文では単に変数を作成しているものと考えることができます．しかしC++では，オブジェクトにコンストラクタがある可能性があるため，変数宣言文によって，大量のプログラムコードが実行される可能性があります．

myclass()関数の定義方法に注目してください．前述のとおり，このコンストラクタ関数には戻り値の型がありません．C++の正式な構文規則では，コンストラクタに戻り値の型を指定するのは誤りです．

グローバルオブジェクトについては，プログラムの実行開始時に，オブジェクトのコンストラクタが一度だけ呼び出されます．ローカルオブジェクトについては，宣言文が実行されるたびにコンストラクタが呼び出されます．

コンストラクタ関数の逆の機能を持つのが，**デストラクタ関数**(destructor function)です．この関数はオブジェクトが破棄されるときに呼び出されます．オブジェクトを扱うときは，オブジェクトを破棄する際に何らかの処理を実行するのが一般的です．たとえば，オブジェクトの作成時にメモリを割り当てた場合は，オブジェクトを破棄する際にメモリを解放します．デストラクタの名前は，それが所属するクラス名の前にチルダ(~)を付けたものになります．次に，デストラクタ関数を含むクラスの例を示します．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(); // コンストラクタ
  ~myclass(); // デストラクタ
  void show();
};

myclass::myclass()
{
  cout << "In constructor¥n";
  a = 10;
}

myclass::~myclass()
```

```
{
  cout << "Destructing...¥n";
}

void myclass::show()
{
  cout << a << "¥n";
}

int main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

クラスのデストラクタは，オブジェクトを破棄するときに呼び出されます．ローカルオブジェクトは，スコープから外れたときに破棄されます．グローバルオブジェクトはプログラムの終了時に破棄されます．

コンストラクタまたはデストラクタのアドレスを取得することはできません．

メモ

技術的に言うと，コンストラクタやデストラクタではあらゆる種類の操作を実行できます．これらの関数内のプログラムコードでは，それらが所属するクラスに関する初期化処理や再設定処理を行わなければならないわけではありません．たとえば，前述の例のコンストラクタでは，$\pi$を100桁まで計算することもできます．しかし，そのオブジェクトの初期化や破棄に直接関係のない処理をコンストラクタやデストラクタで行うのは良いプログラミングスタイルではないので，できるだけ避けてください．

## 例 2.1　コンストラクタ関数とデストラクタ関数

**1.** 第1章で作成したstackクラスでは，スタックの索引値を設定する初期化関数が必要でした．これはまさに，コンストラクタ関数で実行すべき操作です．次に，改良したstackクラスのプログラムコードを示します．このクラスでは，コンストラクタを使ってstackオブジェクトの作成時に初期化を自動的に行います．

```
#include <iostream>
using namespace std;

#define SIZE 10
```

```
// 文字を保存するstackクラスを宣言する
class stack {
  char stck[SIZE];     // スタック領域を確保する
  int tos;             // スタック先頭の索引
public:
  stack();             // コンストラクタ
  void push(char ch);  // スタックに文字をプッシュする
  char pop();          // スタックから文字をポップする
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0; // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
  // 自動的に初期化される2つのスタックを作成する
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
```

```
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "¥n";

  return 0;
}
```

この改良版のクラスでは，独立した関数を明示的に呼び出す必要はなく，初期化処理はコンストラクタ関数によって自動的に実行されます．これは重要です．オブジェクトの作成時に初期化が自動的に行われることによって，手違いで初期化が行われてしまう可能性を排除できます．このことによって，プログラムの複雑さが軽減されます．初期化はオブジェクトの作成時に自動的に行われるので，プログラマは初期化について心配する必要がなくなるのです．

**2.** 次のプログラムでは，コンストラクタとデストラクタの両方を使用しています．このプログラムではstrtypeという名前の簡単な文字列クラスを作成します．このクラスには文字列とその長さを格納します．strtypeオブジェクトの作成時には，文字列を保存するためのメモリを割り当て，その初期の長さを0に設定します．strtypeオブジェクトを破棄する際には，そのメモリを解放します．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

#define SIZE 255

class strtype {
  char *p;
  int len;
public:
  strtype();  // コンストラクタ
  ~strtype(); // デストラクタ
  void set(char *ptr);
  void show();
};

// 文字列オブジェクトを初期化する
strtype::strtype()
{
  p = (char *) malloc(SIZE);
  if(!p) {
```

```
    cout << "Allocation error¥n";
    exit(1);
  }
  *p = '¥0';
  len = 0;
}

// 文字列オブジェクトを破棄する際にメモリを解放する
strtype::~strtype()
{
  cout << "Freeing p¥n";
  free(p);
}

void strtype::set(char *ptr)
{
  if(strlen(ptr) >= SIZE) {
    cout << "String too big¥n";
    return;
  }
  strcpy(p, ptr);
  len = strlen(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1, s2;

  s1.set("This is a test.");
  s2.set("I like C++.");
  s1.show();
  s2.show();

  return 0;
}
```

このプログラムではmalloc()関数とfree()関数を使ってメモリを割り当て，解放しています．これはまったく正当な方法ですが，C++では別の方法でメモリを動的に管理することができます．この方法についてはほかの章で説明します．

メモ

このプログラムでは，Cライブラリ関数に対して新しいスタイルのヘッダを使用しています．第1章で説明したとおり，これらのヘッダをサポートしていないコンパイラを使用する場合は，このヘッダを標準Cヘッダファイルに置き換えてください．本書で紹介するほかのプログラムでも，Cライブラリ関数を使用する場合はすべて同じ処理が必要です．

**3.** 次のプログラムでは，オブジェクトのコンストラクタとデストラクタを面白い方法で利用しています．このプログラムではtimerクラスのオブジェクトを使用して，timer型オブジェクトの作成から破棄までの時間を計測しています．オブジェクトのデストラクタが呼び出されると，経過時間が表示されます．このようなオブジェクトを使用すると，プログラムの実行時間や，ブロック内の関数の実行にかかった時間を計測することができます．時間の計測を終了したいときには，オブジェクトを確実にスコープから外す必要があります．

```
#include <iostream>
#include <ctime>
using namespace std;

class timer {
  clock_t start;
public:
  timer(); // コンストラクタ
  ~timer(); // デストラクタ
};

timer::timer()
{
  start = clock();
}

timer::~timer()
{
  clock_t end;
  end = clock();
  cout << "Elapsed time: " << (end-start) /
                                        ➥CLOCKS_PER_SEC << "¥n";
}

int main()
{
  timer ob;
  char c;
```

```
        // 遅延 ...
        cout << "Press a key followed by ENTER: ";
        cin >> c;

        return 0;
    }
```

このプログラムでは，標準ライブラリ関数のclock()を使用しています．この関数は，プログラムを開始してから発生したクロックサイクルの数を返します．この値をCLOCKS_PER_SECで除算することによって，秒数値に変換することができます．

### 練習問題　2.1　コンストラクタ関数とデストラクタ関数

**1.** 第1章の練習問題で作成したキュークラスを改良し，初期化関数をコンストラクタに置き換えなさい．

**2.** 経過時間を測定するストップウォッチをまねて，stopwatchという名前のクラスを作成しなさい．コンストラクタを使って初期の経過時間を0に設定します．start()とstop()という2つのメンバ関数を作成し，タイマーをそれぞれオンまたはオフに設定します．また，show()というメンバ関数を作成し，経過時間を表示します．さらに，デストラクタ関数を作成して，stopwatchオブジェクトを破棄する際に経過時間を自動的に表示します(単純にするために，時間を秒数で表示します)．

**3.** 次に示すコンストラクタの誤りを指摘しなさい．

```
class sample {
  double a, b, c;
public:
  double sample();  // エラーになる理由は？
};
```

## 2.2　仮引数を受け取るコンストラクタ

コンストラクタ関数には**引数**(argument)を渡すことができます．このためには，コンストラクタ関数の宣言と定義に適切な**仮引数**(parameter)を追加するだけです．そして，そのオブジェクトを宣言するときに引数を指定します．このしくみについて理解するために，短いサンプルプログラムを見てみましょう．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x); // コンストラクタ
  void show();
};

myclass::myclass(int x)
{
  cout << "In constructor¥n";
  a = x;
}

void myclass::show()
{
  cout << a << "¥n";
}

int main()
{
  myclass ob(4);

  ob.show();

  return 0;
}
```

myclassクラスのコンストラクタでは，仮引数を1つ受け取ります．myclass()関数に渡された値を使って，変数aを初期化します．main()関数内でのobの宣言方法に注目してください．obの後に，4という数値が括弧で囲まれて指定されています．これはmyclass()の仮引数xに渡される引数で，変数aを初期化するために使われます．

仮引数を受け取るコンストラクタに引数を渡す際に使っている構文は，実際には次の長い構文を短縮したものです．

```
myclass ob = myclass(4);
```

しかしほとんどのC++プログラマは短い構文を使用します．これらの2つの構文には，コンストラクタをコピーするときに関係する若干の技術的な相違があります．これについては，本書の後の章で説明します．現段階では，この点について心配する必要はありません．

メモ

コンストラクタ関数と異なり，デストラクタ関数は仮引数を受け取ることができません．その理由は簡単です．破棄されるオブジェクトに引数を渡す機構は存在しないからです．

## 例 2.2　仮引数を受け取るコンストラクタ

**1.** コンストラクタには，複数の引数を渡すことができます(実際にはその方が一般的です)．次のmyclass()では，2つの引数を受け取っています．

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int x, int y); // コンストラクタ
  void show();
};

myclass::myclass(int x, int y)
{
  cout << "In constructor¥n";
  a = x;
  b = y;
}

void myclass::show()
{
  cout << a << ' ' << b << "¥n";
}

int main()
{
  myclass ob(4, 7);
  ob.show();
  return 0;
}
```

このプログラムでは，xに4を，yに7を渡しています．これと同じ方法を使って，任意の数の引数を渡すことができます(当然ながら，コンパイラによる制限はあります)．

**2.** 次のプログラムはstackクラスの改良版で，仮引数付きのコンストラクタを使用して，スタックに「名前」を渡しています．この1文字の名前を使って，エラーが発生した場合に参照先のスタックを識別します．

```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字を保存するstackクラスを宣言する
class stack {
  char stck[SIZE];     // スタック領域を確保する
  int tos;             // スタック先頭の索引
  char who;            // スタックを識別する
public:
  stack(char c);       // コンストラクタ
  void push(char ch);  // スタックに文字をプッシュする
  char pop();          // スタックから文字をポップする
};

// スタックを初期化する
stack::stack(char c)
{
  tos = 0;
  who = c;
  cout << "Constructing stack " << who << "¥n";
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack " << who << " is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack " << who << " is empty¥n";
    return 0; // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
```

```
    // 自動的に初期化される2つのスタックを作成する
    stack s1('A'), s2('B');
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    // これによってエラーメッセージが出力される
    for(i=0; i<5; i++) cout << "Pop s1: " << s1.pop() << "¥n";
    for(i=0; i<5; i++) cout << "Pop s2: " << s2.pop() << "¥n";

    return 0;
  }
```

この例のように，オブジェクトに「名前」を与えると，デバッグの際に大変便利です．デバッグの際には，どのオブジェクトによってエラーが発生したかを識別することが重要となります．

**3.** 次に，前に作成したstrtypeクラスを，別の方法で作成する例を示します．このプログラムでは仮引数付きのコンストラクタ関数を使用します．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
```

```
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Freeing p¥n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");
  s1.show();
  s2.show();

  return 0;
}
```

この strtype クラスでは，コンストラクタ関数を使用して文字列の初期値を設定しています．

**4.** 例3のプログラムでは定数を使用していますが，オブジェクトコンストラクタには有効な式(変数を含む)を渡すこともできます．次のプログラムでは，ユーザー入力を使用してオブジェクトを作成しています．

```
#include <iostream>
using namespace std;

class myclass {
  int i, j;
public:
  myclass(int a, int b);
  void show();
};

myclass::myclass(int a, int b)
{
  i = a;
  j = b;
}
```

```
void myclass::show()
{
  cout << i << ' ' << j << "¥n";
}

int main()
{
  int x, y;

  cout << "Enter two integers: ";
  cin >> x >> y;

  // 変数を使用してobを作成する
  myclass ob(x, y);
  ob.show();

  return 0;
}
```

このプログラムは，オブジェクトについて重要な点を示しています．オブジェクトは，作成時の必要性に合わせて正確に作成することができます．C++についてさらに学んでいくと，オブジェクトを「実行時に」作成することの便利さがわかるようになるでしょう．

## 練習問題　2.2　仮引数を受け取るコンストラクタ

**1.** stack クラスを修正し，スタックのメモリを動的に割り当てなさい．コンストラクタ関数への仮引数でスタックのサイズを受け取るようにします(デストラクタ関数でこのメモリを忘れずに解放しなさい)．

**2.** 作成時にコンストラクタで，現在のシステム時刻と日付を仮引数として受け取るt_and_dという名前のクラスを作成しなさい．このクラスに，受け取った時刻と日付を画面に表示するメンバ関数を含めます(時刻と日付を取得および表示するには，標準ライブラリの時刻関数と日付関数を使用します)．

**3.** コンストラクタ関数で，3つの倍精度浮動少数点数を受け取るboxという名前のクラスを作成しなさい．これらの値はそれぞれ立方体の1辺の長さを表します．boxクラスで立方体の体積を計算し，結果を倍精度浮動少数点数変数に格納します．vol()という名前のメンバ関数を作成し，それぞれのboxオブジェクトの体積を表示します．

# 2.3　継承

**継承**(inheritance)については第7章で詳しく説明しますが，ここでも簡単に紹介しておく必要があります．C++では，継承とは1つのクラスがほかのクラスの性質を受け継ぐための機構です．継承によって，最も汎用的なクラスから最も特化されたクラスへの流れを示すクラス階層を作成することができます．

まず，継承について説明する際によく使われる2つの用語について説明しましょう．あるクラスを別のクラスが継承するとき，継承される側のクラスを**基本クラス**(base class)と呼びます．継承する側のクラスを**派生クラス**(derived class)と呼びます．一般に，継承の過程は，基本クラスの定義から始まります．基本クラスは，すべての派生クラスに共通のすべての性質を定義します．基本的に，基本クラスは，最も汎用的な一連の性質を記述します．派生クラスは，これらの汎用的な性質を継承した上で，そのクラスに特有の性質を追加します．

1つのクラスが，ほかのクラスを継承するしくみを理解するために，次の単純な例を見ることにしましょう．この例は単純ですが，継承の多くの主要機能を示しています．

まず，基本クラスの定義を示します．

```
// 基本クラスを定義する
class B {
  int i;
public:
  void set_i(int n);
  int get_i();
};
```

この基本クラスを継承する派生クラスの例を示します．

```
// 派生クラスを定義する
class D : public B {
  int j;
public:
  void set_j(int n);
  int mul();
};
```

この宣言について詳しく説明しましょう．クラス名Dの後に，コロンとpublicキーワード，クラス名Bが続いています．これは，DクラスはBクラスのすべての要素を継承することを示しています．publicキーワードは，Bクラスを継承し，基本クラスのすべての公開要素を派生クラスの公開要素とすることを示します．ただし，基本クラスの非公開要素は非公開なままで，派生クラスから直接アクセスすることはできません．

次に，BクラスとDクラスを使用するプログラム全体のプログラムコードを示します．

```
// 単純な継承の例
#include <iostream>
using namespace std;

// 基本クラスを定義する
class B {
  int i;
public:
  void set_i(int n);
  int get_i();
};

// 派生クラスを定義する
class D : public B {
  int j;
public:
  void set_j(int n);
  int mul();
};

// 基本クラス中の変数iを設定する
void B::set_i(int n)
{
  i = n;
}

// 基本クラス中の変数iの値を返す
int B::get_i()
{
  return i;
}

// 派生クラス中の変数jを設定する
void D::set_j(int n)
{
  j = n;
}

// 基本クラス中の変数iの値に派生クラスの変数jの値を乗じた結果を返す
int D::mul()
{
  // 派生クラスは，基本クラスの公開メンバ関数を呼び出すことができる
  return j * get_i();
}

int main()
{
  D ob;
```

```
    ob.set_i(10); // 基本クラス中のiを設定する
    ob.set_j(4);  // 派生クラス中のjを設定する

    cout << ob.mul();  // 40を表示する

    return 0;
}
```

mul()関数の定義に注目してください．この関数では，特にオブジェクトにリンクすることなく，Dクラスではなく B クラスのメンバである get_i()関数を呼び出しています．B クラスのすべての公開メンバは D クラスの公開メンバとなっているので，このようなことが可能になるのです．ただし mul()関数では変数 i に直接アクセスすることはできないので，get_i()関数を呼び出さなければなりません．これは，基本クラスの非公開メンバ(変数i)は，基本クラスに対して非公開のままであり，どのような派生クラスからもアクセスできないからです．派生クラスから基本クラスの非公開メンバにアクセスできない理由は，カプセル化を維持するためです．クラスを継承するだけで，クラスの非公開メンバが公開になってしまったら，カプセル化は無意味になってしまいます．

次に，基本クラスを継承する一般形式を示します．

継承

```
class derived-class-name : access-specifier base-class-name {
  .
  .
  .
};
```

*access-specifier* には，public，private，protected のいずれかのキーワードを指定します．現段階では，クラスを継承する際には public を使うようにしてください．これらのキーワードの詳細については，本書の後の章で説明します．

## 例 2.3　継承

**1.** 次のプログラムでは，fruit という名前の汎用基本クラスを定義しています．このクラスは，果物が持ついくつかの特性を定義しています．このクラスを，Apple および Orange という名前の 2 つの派生クラスで継承します．これらのクラスは，この種の果物に関する詳細情報を提供しています．

```
// クラス継承の例
#include <iostream>
```

```
#include <cstring>
using namespace std;

enum yn {no, yes};
enum color {red, yellow, green, orange};
void out(enum yn x);
char *c[] = {
  "red", "yellow", "green", "orange"};

// 汎用fruitクラス
class fruit {

// この基本クラスでは，すべての要素が公開
public:
  enum yn annual;
  enum yn perennial;
  enum yn tree;
  enum yn tropical;
  enum color clr;
  char name[40];
};

// Appleクラスを派生する
class Apple : public fruit {
  enum yn cooking;
  enum yn crunchy;
  enum yn eating;
public:
  void seta(char *n, enum color c, enum yn ck, enum yn crchy,
            enum yn e);
  void show();
};

// Orangeクラスを派生する
class Orange : public fruit {
  enum yn juice;
  enum yn sour;
  enum yn eating;
public:
  void seto(char *n, enum color c, enum yn j, enum yn sr,
            enum yn e);
  void show();
};

void Apple::seta(char *n, enum color c, enum yn ck,
                 enum yn crchy, enum yn e)
{
  strcpy(name, n);
  annual = no;
```

```
    perennial = yes;
    tree = yes;
    tropical = no;
    clr = c;
    cooking = ck;
    crunchy = crchy;
    eating = e;
}

void Orange::seto(char *n, enum color c, enum yn j,
                  enum yn sr, enum yn e)
{
    strcpy(name, n);
    annual = no;
    perennial = yes;
    tree = yes;
    tropical = yes;
    clr = c;
    juice = j;
    sour = sr;
    eating = e;
}

void Apple::show()
{
    cout << name << " apple is: " << "¥n";
    cout << "Annual: ";  out(annual);
    cout << "Perennial: ";  out(perennial);
    cout << "Tree: "; out(tree);
    cout << "Tropical: "; out(tropical);
    cout << "Color: " << c[clr] << "¥n";
    cout << "Good for cooking: "; out(cooking);
    cout << "Crunchy: ";  out(crunchy);
    cout << "Good for eating: ";  out(eating);
    cout << "¥n";
}

void Orange::show()
{
    cout << name << " orange is: " << "¥n";
    cout << "Annual: ";  out(annual);
    cout << "Perennial: ";  out(perennial);
    cout << "Tree: ";  out(tree);
    cout << "Tropical: ";  out(tropical);
    cout << "Color: " << c[clr] << "¥n";
    cout << "Good for juice: ";  out(juice);
    cout << "Sour: ";  out(sour);
    cout << "Good for eating: ";  out(eating);
    cout << "¥n";
```

```
}

void out(enum yn x)
{
  if(x==no) cout << "no¥n";
  else cout << "yes¥n";
}

int main()
{
  Apple a1, a2;
  Orange o1, o2;

  a1.seta("Red Delicious", red, no, yes, yes);
  a2.seta("Jonathan", red, yes, no, yes);

  o1.seto("Navel", orange, no, no, yes);
  o2.seto("Valencia", orange, yes, yes, no);

  a1.show();
  a2.show();

  o1.show();
  o2.show();

  return 0;
}
```

基本クラスfruitでは，すべての種類の果物に共通の性質をいくつか定義しています(当然ながら，本書ではこの例を短く収めるために，fruitクラスをある程度簡略化しています)．たとえばすべての果物は，一年生植物か多年生植物に実ります．また，すべての果物は，木かその他の種類の植物(つるや潅木など)に実ります．すべての果物には色と名前があります．この基本クラスをAppleクラスとOrangeクラスが継承しています．これらのクラスは，その果物特有の情報を提供します．

この例は，継承を行う基本的な理由を示しています．ここでは，すべての果物に共通の汎用的な性質を定義する基本クラスを作成しているのです．それぞれの種類の果物特有の性質の定義については，派生クラスに任されています．

このプログラムは，継承に関してもう1つの重要な事実を示しています．基本クラスを1つの派生クラスで独占的に「所有」することはできません．基本クラスを継承できるクラスの数に制限はないのです．

**練習問題 2.3 継承**

1. 次の基本クラスがあります。

```
class area_cl {
public:
  double height;
  double width;
};
```

これを継承して，rectangle（四角形）およびisosceles（二等辺三角形）という名前の2つの派生クラスを作成しなさい．各クラスにarea()という名前の関数を含め，それぞれ四角形または二等辺三角形の面積を返しなさい．仮引数付きのコンストラクタを使用して，height と width を初期化しなさい．

## 2.4　オブジェクトポインタ

これまでは，ドット演算子を使用してオブジェクトのメンバにアクセスしてきました．これは，オブジェクトを操作する正しい方法です．しかし，オブジェクトのメンバにアクセスする際には，そのオブジェクトのポインタを使うこともできます．ポインタを使用するときは，ドット演算子の代わりに**アロー演算子**（**->**）を使用します（構造体へのポインタを使用するときにアロー演算子を使うのとまったく同じです）．

オブジェクトポインタを宣言する方法は，ほかの型へのポインタを宣言する方法と同じです．クラス名を指定し，変数名の前にアスタリスクを挿入します．オブジェクトのアドレスを取得するには，ほかの型の変数のアドレスを取得するときと同様に，オブジェクトの前に & 演算子を挿入します．

ほかの型へのポインタと同様に，オブジェクトポインタをインクリメントすると，その型の次のオブジェクトを参照することができます．

### 例 2.4 オブジェクトポインタ

1. 次のプログラムは，オブジェクトポインタを使用する単純な例です．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
```

```
public:
  myclass(int x); // コンストラクタ
  int get();
};

myclass::myclass(int x)
{
  a = x;
}

int myclass::get()
{
  return a;
}

int main()
{
  myclass ob(120);  // オブジェクトを作成する
  myclass *p;       // オブジェクトへのポインタを作成する

  p = &ob;          // obのアドレスをpに格納する

  cout << "Value using object: " << ob.get();
  cout << "¥n";
  cout << "Value using pointer: " << p->get();

  return 0;
}
```

次の宣言によって，myclass クラスのオブジェクトへのポインタを作成しています．

```
myclass *p;
```

オブジェクトポインタを作成しても，オブジェクトが作成されるわけではないことに注意してください．オブジェクトへのポインタが作成されるだけです．obのアドレスは，次の文によって変数 p に格納されます．

```
p = &ob;
```

また，このプログラムではポインタを使用してオブジェクトのメンバにアクセスしています．

オブジェクトポインタについては，C++についてもう少し学んだ後，第4章で再び取り上げます．第4章では，オブジェクトポインタに関する特殊機能をいくつか紹介します．

## 2.5　クラス，構造体，共用体の関連

これまでに見てきたように，クラスは構文上は構造体と似ています．さらに驚くべきことに，クラスと構造体は機能的にもよく似ています．C++では構造体の定義が拡張され，クラスと同様にメンバ関数，コンストラクタ，デストラクタを含めることができるようになっています．実際，構造体とクラスの違いは，クラスのメンバはデフォルトで非公開であるのに対し，構造体のメンバはデフォルトで公開であるということだけです．次に，構造体の拡張構文を示します．

構造体

```
struct type-name {
    // 公開関数とデータメンバ
private:
    // 非公開関数とデータメンバ
} object-list;
```

C++の正式な構文によると，構造体とクラスはどちらも新しいクラス型を作成します．ここで，新しいキーワードが使われていることに注目してください．privateキーワードは，その後に続くメンバがそのクラスに対して非公開であることを示します．

一見すると，ほとんど同じ機能を持つ構造体とクラスがあることは，無駄であるように思えます．C++の初心者の多くは，なぜこのような重複があるのか不思議に思うようです．実際，classキーワードは不要ではないかという意見を聞くことも少なくありません．

この疑問に対する答えとしては，積極的な理由と消極的な理由の2つがあります．積極的な(避けられない)理由とは，Cからの上方互換性を維持するためです．C++プログラムでは，C形式の構造体も完全にサポートされています．Cではすべての構造体メンバがデフォルトで公開なので，この規則がC++でも維持されています．さらに，classは構文的にstructと異なるので，クラスの定義ではC形式の構造体定義との互換性を考えることなく，クラスを展開することができます．両者が別個のものであれば，互換性の問題によってC++言語の将来が制限される心配はありません．

一方，消極的な理由とは，C++で構造体の定義を拡張してメンバ関数を含めても，特に不利益はないということです．

構造体はクラスと同じ機能を持ちますが，ほとんどのプログラマはC形式の方法で構造体を使用し，構造体にはメンバ関数を含めません．ほとんどのプログラマは，データとプログラムコードの両方を持つオブジェクトを定義する際にはclassキーワードを使用します．ただし，これはスタイル上の問題であり，プログラマの好みによって異なります(本書ではこれ以降，メンバ関数を持たないオブジェクトにはstructキーワードを使用します)．

クラスと構造体の関係に気付いた方は，C++の共用体とクラスにも関係があることにも気が付いたと思います．C++の共用体は，関数とデータの両方をメンバとして含むことのできるクラス型を定義します．共用体は，privateキーワードを使用するまで，デフォルトですべてのメンバが公開であるという点では構造体と似ています．ただし，共用体ではすべてのデータメンバが同じメモリ位置を共有します(Cと同様)．共用体にはコンストラクタ関数とデストラクタ関数を含めることができます．Cの共用体には，C の ++ 共用体との上方互換性があります．

構造体とクラスは，一見無駄のように見えましたが，共用体の場合は違います．オブジェクト指向言語では，カプセル化を維持することが重要です．したがって，プログラムコードとデータをリンクするという共用体の機能によって，すべてのデータが同じメモリ位置を共有するクラス型を作成することができます．クラスではこのようなことはできません．

C++では，共用体にいくつかの制限が課されています．まず，共用体はほかのクラスを継承することができず，どのような型の基本クラスとしても使用できません．共用体ではstaticメンバを持つことはできません．また，コンストラクタやデストラクタを持つオブジェクトを含むことはできません．共用体自体は，コンストラクタとデストラクタを持つことができます．最後に，共用体は仮想メンバ関数を持つことができません(仮想メンバ関数については，本書の後半で説明します)．

C++には，**無名共用体**(anonymous union)と呼ばれる特殊な共用体があります．無名共用体には型名がなく，この種の共用体では変数を宣言することができません．代わりに，無名共用体ではコンパイラに対して，メンバが同じメモリ位置を共有することを伝えます．ただし，これ以外の面では，メンバは通常の変数と同じように動作し，同じように扱われます．つまり，ドット演算子構文を使用せずに，メンバに直接アクセスすることができます．例として，次のプログラムコードを見てみましょう．

```
union { // 無名共用体
  int i;
  char ch[4];
} ;

i = 10; // 変数iにアクセスし，ディレクトリを変更する
ch[0] = 'X';
```

このように，iとchはどのオブジェクトにも属さないので，直接アクセスすることができます．ただし，これらは同じメモリ領域を共有します．

無名共用体を使用する理由は，2つ以上の変数で同じメモリ位置を共有することを，簡単にコンパイラに伝えることができるということです．この特別な性質を除けば，無名共用体のメンバはほかの変数と同じように動作します．

無名共用体は，通常の共用体に課されているすべての制限に加え，いくつかの制限が追加されています．グローバル無名共用体は，staticとして宣言しなければなりません．無名共用体には，非公開メン

バを含めることはできません．無名共用体のメンバの名前は，同じスコープ内のほかの識別子と重複してはいけません．

## 例 2.5　クラス，構造体，共用体の関連

**1.** 次の短いプログラムでは，struct を使用してクラスを作成しています．

```
#include <iostream>
#include <cstring>
using namespace std;

// structを使用してクラス型を定義する
struct st_type {
  st_type(double b, char *n);
  void show();
private:
  double balance;
  char name[40];
} ;

st_type::st_type(double b, char *n)
{
  balance = b;
  strcpy(name, n);
}

void st_type::show()
{
  cout << "Name: " << name;
  cout << ": $" << balance;
  if(balance<0.0) cout << "***";
  cout << "¥n";
}

int main()
{
  st_type acc1(100.12, "Johnson");
  st_type acc2(-12.34, "Hedricks");

  acc1.show();
  acc2.show();

  return 0;
}
```

前述のとおり，構造体のメンバはデフォルトではすべて公開です．非公開メンバを宣言するには，private キーワードを使う必要があります．

また，C形式の構造体とC++形式の構造体には相違が1つあります．C++では，構造体のタグ名が，そのオブジェクトを宣言するときに使用する完全な型名となります．それに対してCでは，タグ名を完全な型名とするには，その前にstructキーワードを付けなければなりません．

次のプログラムは，クラスを使用して同じプログラムを書き直したものです．

```
#include <iostream>
#include <cstring>
using namespace std;

class cl_type {
  double balance;
  char name[40];
public:
  cl_type(double b, char *n);
  void show();
} ;

cl_type::cl_type(double b, char *n)
{
  balance = b;
  strcpy(name, n);
}

void cl_type::show()
{
  cout << "Name: " << name;
  cout << ": $" << balance;
  if(balance<0.0) cout << "***";
  cout << "¥n";
}

int main()
{
  cl_type acc1(100.12, "Johnson");
  cl_type acc2(-12.34, "Hedricks");

  acc1.show();
  acc2.show();

  return 0;
}
```

**2.** 次のプログラムでは，共用体を使用して，double値に含まれるバイナリビットパターンをバイト単位で表示しています．

```
#include <iostream>
using namespace std;

union bits {
  bits(double n);
  void show_bits();
  double d;
  unsigned char c[sizeof(double)];
};

bits::bits(double n)
{
  d = n;
}

void bits::show_bits()
{
  int i, j;
  for(j = sizeof(double)-1; j>=0; j--) {
    cout << "Bit pattern in byte " << j << ": ";
    for(i = 128; i; i >>= 1)
      if(i & c[j]) cout << "1";
      else cout << "0";
    cout << "¥n";
  }
}

int main()
{
  bits ob(1991.829);
  ob.show_bits();
  return 0;
}
```

このプログラムからの出力を次に示します．

```
Bit pattern in byte 7: 01000000
Bit pattern in byte 6: 10011111
Bit pattern in byte 5: 00011111
Bit pattern in byte 4: 01010000
Bit pattern in byte 3: 11100101
Bit pattern in byte 2: 01100000
Bit pattern in byte 1: 01000001
Bit pattern in byte 0: 10001001
```

**3.** 構造体と共用体はどちらもコンストラクタとデストラクタを持つことができます．次に示すのは，strtype クラスを構造体として書き直したものです．このプログラムでは，コンストラクタ関数とデストラクタ関数を両方とも使用しています．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

struct strtype {
  strtype(char *ptr);
  ~strtype();
  void show();
private:
  char *p;
  int len;
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Freeing p¥n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();
```

```
  return 0;
}
```

**4.** 次のプログラムでは，無名共用体を使用して，double 値を構成する個々のバイトを表示しています(このプログラムでは，double 値の長さを 8 バイトと想定しています).

```
// 無名共用体を使用する
#include <iostream>
using namespace std;

int main()
{
  union {
    unsigned char bytes[8];
    double value;
  };
  int i;
  value = 859345.324;

  // double値内のバイトを表示する
  for(i=0; i<8; i++)
    cout << (int) bytes[i] << " ";

  return 0;
}
```

ご覧のように，value と bytes には，共用体の一部ではない通常の変数とまったく同じようにアクセスすることができます．これらは無名共用体の一部として宣言されていますが，これらの名前は，同じ箇所で宣言されているほかのローカル変数と同じスコープレベルを持ちます．これが，無名共用体のメンバがそのスコープ内にあるほかの変数と同じ名前を使用できない理由です．

## 練習問題　2.5　クラス，構造体，共用体の関連

**1.** 2.1 節で作成した stack クラスを修正し，クラスではなく構造体を使用しなさい．

**2.** 共用体クラスを使用して，整数の下位バイトと上位バイトを入れ換えなさい(16 ビット整数を仮定しているので，32 ビットのコンピュータを使用する場合は，short int 値のバイトを入れ換えなさい)．

**3.** 無名共用体とは何か，通常の共用体とどのように異なるかを説明しなさい．

## 2.6　インライン関数

クラスの説明を先に進める前に，関連する事柄について説明しなければなりません．C++では，実際に呼び出す代わりに，各呼び出しごとにインラインで展開する関数を定義することができます．これは，Cの仮引数付きマクロのしくみとよく似ています．**インライン関数**(inline function)の利点は，関数の呼び出しと終了に伴うオーバーヘッドが発生しないということです．つまり，インライン関数は通常の関数よりもはるかに速く実行することができるのです(関数の呼び出しと終了を行うためのマシン命令によって，関数を実行するたびに時間が消費されます．仮引数があると，さらに多くのオーバーヘッドが発生します)．

インライン関数の欠点としては，インライン関数が大きく，何度も呼び出すと，プログラム自体が大きくなるということです．このため，一般には短い関数だけをインライン関数として宣言します．

インライン関数を呼び出すには，関数定義の前にinline指定子を付けるだけです．次の短いプログラムは，インライン関数の定義方法を示しています．

```
// インライン関数の例
#include <iostream>
using namespace std;

inline int even(int x)
{
  return !(x%2);
}

int main()
{
  if(even(10)) cout << "10 is even¥n";
  if(even(11)) cout << "11 is even¥n";

  return 0;
}
```

この例のeven()関数は，インライン関数として定義されており，引数が偶数である場合に真を返します．つまり，次の2つの行は，機能的には同等です．

```
if(even(10))  cout << "10 is even¥n";

if(!(10%2)) cout << "10 is even¥n";
```

この例から，inlineを使う際に重要となるもう1つの事柄が明らかになっています．インライン関数は，最初に使用する前に定義しなければなりません．そうしないと，コンパイラはインラインで展開することを認識できません．このため，even()関数はmain()関数よりも前に定義されています．

仮引数付きマクロではなくinlineを使うことの利点は2つあります．第1に，より構造的な方法で，短い関数をインラインで展開できることです．たとえば，仮引数付きマクロを作成する際には，インライン展開を毎回正しく行うためには，括弧を忘れずに付けなければなりません．インライン関数を使えば，そのような問題は起こりません．

第2の理由は，コンパイラは，インライン関数を，マクロ展開よりも徹底して最適化を行うことができるということです．どのような場合にも，C++ プログラマはほとんど仮引数付きマクロを使用せずに，inline を使用することによって，短い関数の呼び出しに伴うオーバーヘッドを防ぎます．

inline 指定子は，コンパイラにとってはコマンドではなく要求であることを覚えておいてください．何らかの理由でコンパイラが要求を実行できない場合，関数は通常の関数としてコンパイルされ，inline 要求は無視されます．

コンパイラによっては，インライン関数にいくつかの制限が課されます．たとえば，static変数やループ文，switch文やgoto文，再帰を含む関数は，インライン化されないコンパイラもあります．各コンパイラにおけるインライン関数の制限については，使用するマニュアルのユーザーマニュアルを参照してください．

**重要** インライン関数に関する条件が満たされない場合，コンパイラは通常の関数としてコンパイラを生成します．

## 例 2.6 インライン関数

1. 関数は，クラスのメンバ関数を含め，どのようなものでもインライン化することができます．次に示すメンバ関数divisible()は，実行速度を上げるために，インライン関数として定義されています(この関数は，1つ目の引数が2つ目の引数によって割り切れる場合は，真を返します)．

```
// インラインメンバ関数の例
#include <iostream>
using namespace std;

class samp {
  int i, j;
public:
  samp(int a, int b);
  int divisible(); // 定義でインライン化する
};

samp::samp(int a, int b)
```

```
{
  i = a;
  j = b;
}

/* iがjによって割り切れる場合に1を返す
   このメンバ関数はインラインで展開される
*/
inline int samp::divisible()
{
  return !(i%j);
}

int main()
{
  samp ob1(10, 2), ob2(10, 3);

  // これは真
  if(ob1.divisible()) cout << "10 divisible by 2¥n";

  // これは偽
  if(ob2.divisible()) cout << "10 divisible by 3¥n";

  return 0;
}
```

**2.** オーバーロード関数をインライン化することもできます．次のプログラムでは，min()関数を３とおりにオーバーロードしています．各関数はそれぞれinlineを使用して宣言されています．

```
#include <iostream>
using namespace std;

// min()関数を3とおりにオーバーロードする
// 整数
inline int min(int a, int b)
{
  return a<b ? a : b;
}

// 長整数
inline long min(long a, long b)
{
  return a<b ? a : b;
}

// 倍精度浮動少数点数
inline double min(double a, double b)
```

```
{
  return a<b ? a : b;
}

int main()
{
  cout << min(-10, 10) << "¥n";
  cout << min(-10.01, 100.002) << "¥n";
  cout << min(-10L, 12L) << "¥n";

  return 0;
}
```

### 練習問題 2.6 インライン関数

1. 第1章ではabs()関数をオーバーロードし，整数，長整数，倍精度浮動少数点数の絶対値を求めるようにしました．このプログラムを修正し，これらの関数をインライン関数として展開しなさい．

2. 次の関数は，コンパイル時にインライン化されません．その理由を説明しなさい．

```
void f1()
{
  int i;
  for(i=0; i<10; i++) cout << i;
}
```

## 2.7 自動インライン化

メンバ関数の定義が短い場合は，クラス定義の中にその定義を含めることができます．このようにした場合，関数は可能であれば自動的にインライン化されます．クラス定義内で関数を定義する場合，inline キーワードは必要ありません(ただし，inline キーワードを使用してもエラーにはなりません)．たとえば，前の節で説明したdivisible()関数を自動インライン化するには，次のようにします．

```
#include <iostream>
using namespace std;

class samp {
  int i, j;
public:
  samp(int a, int b);
```

```
    /* divisible()をここで定義すると，
       自動的にインライン化される */
    int divisible() { return !(i%j); }
};

samp::samp(int a, int b)
{
  i = a;
  j = b;
}

int main()
{
  samp ob1(10, 2), ob2(10, 3);

  // これは真
  if(ob1.divisible()) cout << "10 divisible by 2¥n";

  // これは偽
  if(ob2.divisible()) cout << "10 divisible by 3¥n";

  return 0;
}
```

divisible()関数のプログラムコードは，samp クラスの定義内に記述されています．divisible()関数をこれ以外に定義する必要はなく，また定義してはいけません．sampクラス内でdivisible()関数を定義すると，自動的にインライン関数となります．

クラス定義内で定義された関数をインライン化できない場合（インライン関数の条件が満たされないため）は，自動的に通常の関数としてコンパイルされます．

samp クラス内での divisible()関数の定義の本体に注目してください．本体は１行ですべて定義されています．C++プログラムでは，関数をクラス宣言内で宣言する場合，この形式をよく使用します．こうすることによって，宣言がより簡潔になります．ただし，sampクラスを次のように書くこともできます．

```
class samp {
  int i, j;
public:
  samp(int a, int b);

  /* divisible()をここで定義すると，
     自動的にインライン化される */
  int divisible()
  {
    return !(i%j);
  }
};
```

このプログラムでは，標準的なインデントスタイルを使用してdivisible()関数を記述しています．コンパイラの視点から見ると，前述の簡潔なスタイルとこの標準スタイルに違いはありません．しかし，C++プログラムで，クラス定義内に短い関数を定義する際には，簡潔なスタイルの方がよく使われます．

クラス宣言内の自動インライン関数にも，「通常の」インライン関数に課されるのと同じ制限が当てはまります．

## 例 2.7 自動インライン化

**1.** クラス内でインライン関数を定義する最も一般的な用途は，コンストラクタ関数とデストラクタ関数の定義です．次の例では，sampクラスをより効率的に定義しています．

```
#include <iostream>
using namespace std;

class samp {
  int i, j;
public:
  // インラインコンストラクタ
  samp(int a, int b) { i = a; j = b; }
  int divisible() { return !(i%j); }
};
```

sampクラス内でsamp()関数を定義しているので，samp()関数を別個に定義する必要はありません．

**2.** 自動インライン化機能の効果が低いか，またはないときでも，クラス宣言に短い関数が含まれることがあります．次の宣言について考えてみましょう．

```
class myclass {
  int i;
public:
  myclass(int n) { i = n; }
  void show() { cout << i; }
};
```

このshow()関数は，自動的にインライン関数になっています．しかし，入出力演算子は一般にはCPU/メモリ操作に比べて低速なので，関数を呼び出すオーバーヘッドを取り除いた効果は，事実上得られません．このような場合にも特に害はないので，C++プログラムでは，単に便宜上の理由から，このような小さな関数をクラス内で宣言することがよくあります．

### 練習問題　2.7　自動インライン化

1. 2.1節の例1で作成したstackクラスを修正し，適切な場所で自動インライン化関数を使用しなさい．

2. 2.2節の例3で作成したstrtypeクラスを修正し，自動インライン化関数を使用しなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. コンストラクタとデストラクタとは何か，いつ実行されるかを説明しなさい．

2. 画面上に線を描くlineという名前のクラスを作成し，lenという名前の非公開整数変数に線の長さを保存します．lineクラスのコンストラクタでは，線の長さを仮引数として受け取ります．コンストラクタでは線の長さを保存し，線を実際に描画します．グラフィックスがサポートされていないシステムの場合は，*を使用して線を描きます．また，余裕があれば，lineクラスのデストラクタを作成し，線を消去しなさい．

3. 次のプログラムによって何が表示されるか考えなさい．

```
#include <iostream>
using namespace std;

int main()
{
  int i = 10;
  long l = 1000000;
  double d = -0.0009;

  cout << i << ' ' << l << ' ' << d;
  cout << "¥n";

  return 0;
}
```

4. 2.3節の練習問題1で作成したarea_clクラスを継承する，もう1つの派生クラスを作成しなさい．このクラスにcylinder（シリンダ）という名前を付け，シリンダの表面積

を計算します．シリンダの表面積は，「$2 \times \pi \times R^2 + \pi \times D \times$ 高さ」という式で算出します．

**5.** インライン関数とは何か説明しなさい．また，その利点と欠点について説明しなさい．

**6.** 次のプログラムを修正し，すべてのメンバ関数を自動的にインライン化しなさい．

```
#include <iostream>
using namespace std;

class myclass {
  int i, j;
public:
  myclass(int x, int y);
  void show();
};

myclass::myclass(int x, int y)
{
  i = x;
  j = y;
}

void myclass::show()
{
  cout << i << " " << j << "¥n";
}

int main()
{
  myclass count(2, 3);

  count.show();

  return 0;
}
```

**7.** クラスと構造体の違いを説明しなさい．

**8.** 次のプログラムコードが有効かどうか考えなさい．

```
union {
  float f;
  unsigned int bits;
} ;
```

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** prompt という名前のクラスを作成し，コンストラクタ関数に任意のプロンプト文字列を渡します．コンストラクタではその文字列を表示し，入力として整数を受け取ります．この値をcountという名前の非公開変数に保存します．prompt型のオブジェクトを破棄する際には，ユーザーが入力した回数だけ，コンピュータのビープ音を鳴らします．

**2.** 第１章では，フィート数をインチ数に変換するプログラムを作成しました．ここで，同じ処理を行うクラスを作成しなさい．このクラスでは，フィート数とそれに対応するインチ数を保存します．クラスのコンストラクタにフィート数を渡し，コンストラクタではインチ数を表示します．

**3.** 1つの非公開整数変数を含むdiceという名前のクラスを作成しなさい．また，標準乱数ジェネレータであるrand()関数を使用して１から６までの範囲の整数を生成するroll()という関数を作成しなさい．roll()関数ではその乱数を表示します．

# 3

# クラスの詳細

## この章の内容

この章では，クラスについてさらに学習を続けます．オブジェクトの割り当て方法，関数にオブジェクトを渡す方法，関数からのオブジェクトを返す方法について学びます．また，フレンド関数という新しい重要な関数についても学びます．

## 前章の理解度チェック

この章を読み進む前に，次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

**1.** 次のクラスがあります．このクラスのコンストラクタ関数とデストラクタ関数の名前を答えなさい．

```
class widgit {
  int x, y;
public:
  // ... コンストラクタ関数とデストラクタ関数を記述する
};
```

**2.** コンストラクタ関数とデストラクタ関数が，それぞれいつ呼び出されるか説明しなさい．

**3.** 次の基本クラスがあります．このクラスをMarsという名前の派生クラスによって継承しなさい．

```
class planet {
  int moons;
  double dist_from_sun;
  double diameter;
  double mass;
public:
 // ...
};
```

**4.** 関数をインラインに展開させる２つの方法を説明しなさい．

**5.** インライン関数の制限を２つ説明しなさい．

**6.** 次のクラスがあります．obという名前のオブジェクトを宣言し，変数aに100，変数cにXを代入しなさい．

```
class sample {
  int a;
  char c;
public:
  sample(int x, char ch) { a = x; c = ch; }
  // ...
};
```

## 3.1　オブジェクトの代入

2つのオブジェクトの型が同じであれば，1つのオブジェクトをもう1つのオブジェクトに代入することができます．デフォルトでは，1つのオブジェクトをもう1つのオブジェクトに代入すると，すべてのデータメンバがビット単位でコピーされます．たとえば，o1という名前のオブジェクトをo2という名前のほかのオブジェクトに代入すると，o1のデータの内容はすべてo2の該当するメンバにコピーされます．次に例を示します．

```
// オブジェクト代入の例
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "¥n"; }
};

int main()
{
  myclass o1, o2;

  o1.set(10, 4);

  // o1をo2に代入する
  o2 = o1;

  o1.show();
  o2.show();

  return 0;
}
```

このプログラムでは，オブジェクトo1のメンバ変数aとbをそれぞれ10と4に設定しています．次に，o1オブジェクトをo2オブジェクトに代入します．これによって，o1.aの現在値がo2.aに，o1.bの現在値がo2.bに代入されます．したがって，このプログラムを実行すると，次のように表示されます．

```
10 4
10 4
```

2つのオブジェクト間で代入を行うと，両者のデータが単に同じになるだけです．2つのオブジェクトは依然としてまったく別個のものです．たとえば，代入の後，o1.set()関数を呼び出して変数o1.aの値を変更しても，o2オブジェクトやその変数値には何の影響もありません．

## 例 3.1　オブジェクトの代入

**1.** 代入文では同じ型のオブジェクトしか使用できません．2つのオブジェクトの型が異なると，コンパイル時にエラーが発生します．また，2つの型は物理的に似ているだけでは不十分で，型名が同じでなければなりません．たとえば，次のプログラムは正しくありません．

```
// このプログラムには誤りがある
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "¥n"; }
};

/* このクラスはmyclassに似ているが，
   クラス名が異なるので，
   コンパイラにとってはまったく別個の型のクラスである
*/
class yourclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "¥n"; }
};

int main()
{
  myclass o1;
```

```
  yourclass o2;

  o1.set(10, 4);

  o2 = o1; // エラー．オブジェクトの型が異なる

  o1.show();
  o2.show();

  return 0;
}
```

この場合，myclassクラスとyourclassクラスは物理的には同じですが，型名が異なるので，コンパイル時には別の型として扱われます．

**2.** 代入を行うと，1つのオブジェクトのすべてのデータメンバがもう1つのオブジェクトに代入されます．これには，配列などの複合データも含まれます．たとえば，次のstackプログラムでは，実際に文字をプッシュしているのはs1だけです．しかし代入を行っているので，s2のstck配列にも文字a，b，cが格納されます．

```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字を格納するstackクラスを宣言する
class stack {
  char stck[SIZE];      // スタック領域を確保する
  int tos;              // スタックの先頭の索引
public:
  stack();              // コンストラクタ
  void push(char ch);   // スタックに文字をプッシュする
  char pop();           // スタックから文字をポップする
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
```

```
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0; // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
  // 自動的に初期化される2つのスタックを作成する
  stack s1, s2;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  // clone s1
  s2 = s1;  // これで，s1とs2は同じになる

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "¥n";

  return 0;
}
```

**3.** オブジェクトをほかのオブジェクトに代入する際には注意が必要です．たとえば，次に示すプログラムは，第2章で作成したstrtypeクラスに短いmain()関数を追加したものです．このプログラムには問題点がありますが，どの部分が問題か考えてみてください．

```
// このプログラムにはエラーがある
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
```

```
class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Freeing p¥n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  // s1をs2に代入する．これによってエラーが発生する
  s2 = s1;

  s1.show();
  s2.show();

  return 0;
}
```

このプログラムの問題点は非常にわかりにくくなっています．s1とs2を作成するとき，どちらもそれぞれの文字列を格納するためのメモリを割り当てます．各オブジェクトに割り当てられたメモリのポインタはpに保存されます．strtypeオブジェクトを破棄する際にはこのメモリが解放されます．しかしs1をs2に代入すると，s2の変数pは，s1の変数pと同じメモリを指すようになります．したがってこれらのオブジェクトを破棄すると，s1の変数pが指すメモリは2度解放されることになり，s2の変数pが最初に指していたメモリはまったく解放されません．

実際のプログラムでこの種の問題が起こると，動的割り当てシステムでエラーが起こります．さらに，プログラムがクラッシュすることもあります．この例からもわかるとおり，オブジェクトを代入する場合は，後から必要となるデータを破棄することがないように，十分に注意しなければなりません．

## 練習問題　3.1　オブジェクトの代入

**1.**　次のプログラムの誤りを指摘しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
using namespace std;

class cl1 {
  int i, j;
public:
  cl1(int a, int b) { i = a; j = b; }
  // ...
};

class cl2 {
  int i, j;
public:
  cl2(int a, int b) { i = a; j = b; }
  // ...
};

int main()
{
  cl1 x(10, 20);
  cl2 y(0, 0);
  x = y;
```

```
        // ...
    }
```

**2.** 第2章の2.1節の練習問題1で作成したqueueクラスを使用して，1つのキューを別のキューに代入するプログラムを作成しなさい．

**3.** 練習問題2のqueueクラスで，キューを格納するメモリを動的に割り当てたとします．この場合，キューを別のキューに代入してはいけない理由を説明しなさい．

# 3.2　関数へのオブジェクトの引き渡し

関数に引数としてデータを渡すのと同じように，オブジェクトを引数として渡すこともできます．このためには，渡すクラスの型を使用して関数の仮引数を宣言するだけです．ほかの型のデータと同様に，デフォルトではすべてのオブジェクトが**値呼び出し**(pass by value)によって関数に渡されます．

## 例　3.2　関数へのオブジェクトの引き渡し

**1.** 次の短いプログラムでは，関数にオブジェクトを渡しています．

```
#include <iostream>
using namespace std;

class samp {
  int i;
public:
  samp(int n) { i = n; }
  int get_i() { return i; }
};

// o.iの2乗を返す
int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

int main()
{
  samp a(10), b(2);

  cout << sqr_it(a) << "¥n";
```

```
    cout << sqr_it(b) << "¥n";

    return 0;
  }
```

このプログラムでは，sampというクラスを作成しています．このクラスはiという1つの整数変数を含みます．sqr_it()関数はsamp型の引数を受け取り，そのオブジェクトのi値の2乗を返します．このプログラムからの出力は，100と4になります．

**2.** 前述のとおり，C++におけるデフォルトの仮引数引き渡し方法は値呼び出しです．つまり，引数がビット単位でコピーされ，関数ではこのコピーが使われます．したがって，関数の内部でオブジェクトに変更を加えても，呼び出し側のオブジェクトには影響ありません．次のプログラムに例を示します．

```
/*
   オブジェクトは，ほかの仮引数と同様，値によって渡される
   したがって，関数の内部で仮引数に変更を加えても，
   呼び出し時に使用したオブジェクト自体には何の影響もない
*/
#include <iostream>
using namespace std;

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* o.iをその2乗に設定する
   ただし，sqr_it()を呼び出す際に使われた
   オブジェクトには何の影響もない
*/
void sqr_it(samp o)
{
  o.set_i(o.get_i() * o.get_i());

  cout << "Copy of a has i value of " << o.get_i();
  cout << "¥n";
}

int main()
{
  samp a(10);
  sqr_it(a); // 値呼び出し
```

```
  cout << "But, a.i is unchanged in main: ";
  cout << a.get_i();  // 10を表示する

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
Copy of a has i value of 100
But, a.i is unchanged in main: 10
```

**3.** ほかの型の変数と同様に，オブジェクトのアドレスを関数に渡すと，関数を呼び出す際に使用した引数を関数内で修正することができます．たとえば，次に示すのは，前の例に示したプログラムと似ていますが，sqr_it()関数にオブジェクトのアドレスを渡しているので，オブジェクトの値が修正されます．

```
/*
   このプログラムでは，sqr_it()関数にオブジェクトのアドレスを渡す
   関数を呼び出す際にオブジェクトのアドレスを渡すと，
   関数内でオブジェクトの値を修正することができる
*/
#include <iostream>
using namespace std;

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* o.iをその2乗に設定する
   sqr_it()関数を呼び出す際に使われた
   オブジェクトに影響する
*/
void sqr_it(samp *o)
{
   o->set_i(o->get_i() * o->get_i());
  cout << "Copy of a has i value of " << o->get_i();
  cout << "¥n";
}

int main()
{
  samp a(10);
```

```
    sqr_it(&a); // aのアドレスをsqr_it()関数に渡す

    cout << "Now, a in main() has been changed: ";
    cout << a.get_i();  // 100を表示する

    return 0;
}
```

このプログラムからの出力は次のようになります．

```
Copy of a has i value of 100
Now, a in main() has been changed: 100
```

**4.** 関数を呼び出す際にオブジェクトをコピーすると，新しいオブジェクトが作成されることになります．またオブジェクトを渡した関数が終了すると，引数として渡したコピーは破棄されます．これによって2つの疑問が生じます．まずコピーを作成する際に，コンストラクタは呼び出されるのでしょうか．また，コピーを破棄する際に，デストラクタは呼び出されるのでしょうか．その答えは，一見意外なものです．

関数呼び出しで使用するオブジェクトのコピーを作成するとき，コンストラクタ関数は呼び出されません．その理由は，考えてみれば簡単にわかります．コンストラクタ関数は，一般にはオブジェクトの何かを初期化するためのものなので，すでに存在するオブジェクトを関数に渡すためにコピーするときには，呼び出す必要がありません．コンストラクタを呼び出すとオブジェクトの内容が変わってしまいます．関数にオブジェクトを渡すときは，初期状態ではなく現在の状態でオブジェクトを渡す必要があります．

ただし，関数が終了してオブジェクトのコピーが破棄されるときには，デストラクタ関数が呼び出されます．これは，オブジェクトがスコープから外れるときに，実行しなければならない処理があるかもしれないからです．たとえば，オブジェクトのコピーでメモリが割り当てられており，それを解放しなければならないこともあります．

まとめると，関数の引数として使うオブジェクトのコピーを作成するときには，コンストラクタ関数は呼び出されません．しかし，オブジェクトのコピーを破棄するとき(通常は，関数が返し，コピーがスコープから外れるとき)には，デストラクタ関数が呼び出されます．

次のプログラムは，この説明を例示しています．

```
#include <iostream>
using namespace std;
```

```
class samp {
  int i;
public:
  samp(int n) {
    i = n;
    cout << "Constructing¥n";
  }
  ~samp() { cout << "Destructing¥n"; }
  int get_i() { return i; }
};

// o.iの2乗を返す
int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

int main()
{
  samp a(10);
  cout << sqr_it(a) << "¥n";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
Constructing
Destructing
100
Destructing
```

ご覧のとおり，コンストラクタ関数の呼び出しは1回しか呼び出されていません．これはaオブジェクトの作成時です．しかし，デストラクタ関数は2回呼び出されています．1つはsqr_it()関数に渡したコピー用のデストラクタであり，もう1つはaオブジェクト自体のデストラクタです【監注：コンパイラによっては，関数を呼ぶ側と呼ばれる側で二重にオブジェクトを複製するので，それらを破棄するためにデストラクタが3回呼び出されることがあります．g++は，そのようなコンパイラの一例です】．

関数の終了時に，引数のコピーオブジェクトのデストラクタが実行されることによって，問題が発生することがあります．たとえば，引数として使用したオブジェクトが，動的にメモリを割り当て，破棄されるときにそのメモリを解放する場合，このオブジェクトのコピーはデストラクタの中で同じメモリを解放することになります．これによって元のオブジェクトが破壊され，実質的に使用できなくなります(次の練習問題

2を参照)．この種のエラーには十分に備え，引数に使用したオブジェクトのコピーのデストラクタ関数によって，元のオブジェクトに変更が及ぶようなことがないように注意しなければなりません．

仮引数として渡したコピーオブジェクトのデストラクタ関数によって，元のオブジェクトが使用するデータが破壊されてしまうという問題の対処法の1つとして，オブジェクト自体ではなくオブジェクトのアドレスを渡す方法があります．アドレスを渡す場合は，新しいオブジェクトが作成されることはないので，関数の終了時にデストラクタが呼び出されることもありません(次の章で説明しますが，C++には非常に優れた別の対処法が用意されています)．ただし，より優れた対処法も存在します．この対処法は，コピーコンストラクタと呼ばれる特殊なコンストラクタについて学んだ後に使用します(コピーコンストラクタについては，第5章で説明します)．

## 練習問題　3.2　関数へのオブジェクトの引き渡し

**1.** 3.1節の例2で作成したstackプログラムを使用し，showstack()という関数を追加しなさい．この関数にはstack型のオブジェクトを渡します．この関数内で，スタックの内容を表示しなさい．

**2.** 関数にオブジェクトを渡すと，そのオブジェクトのコピーが作成されます．さらに，その関数が終了すると，オブジェクトのコピーのデストラクタ関数が呼び出されます．このことを踏まえて，次のプログラムの誤りを指摘しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
#include <cstdlib>
using namespace std;

class dyna {
  int *p;
public:
  dyna(int i);
  ~dyna() { free(p); cout << "freeing ¥n"; }
  int get() { return *p; }
};

dyna::dyna(int i)
{
  p = (int *) malloc(sizeof(int));
```

```
    if(!p) {
      cout << "Allocation failure¥n";
      exit(1);
    }
    *p = i;
  }

  // *ob.pの負の値を返す
  int neg(dyna ob)
  {
    return -ob.get();
  }

  int main()
  {
    dyna o(-10);

    cout << o.get() << "¥n";
    cout << neg(o) << "¥n";

    dyna o2(20);
    cout << o2.get() << "¥n";
    cout << neg(o2) << "¥n";

    cout << o.get() << "¥n";
    cout << neg(o) << "¥n";

    return 0;
  }
```

## 3.3　関数からのオブジェクトの返し

関数にオブジェクトを渡すことができるのと同じように，関数からオブジェクトを返すことができます．そのためには，まず返すクラスの型を使用して関数を宣言します．次に，通常のreturn 文を使用して，その型のオブジェクトを返します．

関数からオブジェクトを返す方法について理解するためには，重要なポイントが1つあります．関数からオブジェクトを返すときには，戻り値を格納する一時オブジェクトが自動的に作成されます．関数からはこのオブジェクトが返されます．値を返した後，このオブジェクトは破棄されます．この一時オブジェクトを破棄することによって，予期しなかった副作用が生じることがあります．次の例2ではこの問題を示しています．

## 例 3.3 関数からのオブジェクトの返し

**1.** 次のプログラムは，オブジェクトを返す関数の例です．

```
// オブジェクトを返す
#include <iostream>
#include <cstring>
using namespace std;

class samp {
  char s[80];
public:
  void show() { cout << s << "¥n"; }
  void set(char *str) { strcpy(s, str); }
};

// samp型のオブジェクトを返す
samp input()
{
  char s[80];
  samp str;

  cout << "Enter a string: ";
  cin >> s;

  str.set(s);

  return str;
}

int main()
{
  samp ob;

  // 返されたオブジェクトをobに代入する
  ob = input();
  ob.show();

  return 0;
}
```

この例のinput()関数では，strという名前のローカルオブジェクトを作成し，キーボードから文字列を読み取ります．この文字列を変数str.sにコピーし，strオブジェクトを関数から返します．このオブジェクトは，main()関数内でinput()関数から返されたときに，obに代入されます．

**2.** 関数からオブジェクトを返す場合，そのオブジェクトにデストラクタ関数が含まれる場合は注意が必要です．返されるオブジェクトは，呼び出し元ルーチンに値を返すとすぐにスコープから外れるからです．たとえば，関数から返されたオブジェクトに，動的に割り当てたメモリを解放するデストラクタがある場合，戻り値を代入したオブジェクトがまだメモリを使用している場合でも，そのメモリは解放されます．たとえば，次のプログラムは前の例のプログラムと似ていますが，誤りがあります．

```
// オブジェクトを返すことによってエラーが生成される
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class samp {
  char *s;
public:
  samp() { s = '¥0'; }
  ~samp() { if(s) free(s); cout << "Freeing s¥n"; }
  void show() { cout << s << "¥n"; }
  void set(char *str);
};

// 文字列をロードする
void samp::set(char *str)
{
  s = (char *) malloc(strlen(str)+1);
  if(!s) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(s, str);
}

// samp型のオブジェクトを返す
samp input()
{
  char s[80];
  samp str;

  cout << "Enter a string: ";
  cin >> s;

  str.set(s);
  return str;
}
```

```
int main()
{
  samp ob;

  // 返されたオブジェクトをobに代入する
  ob = input();   // ここでエラーが発生する
  ob.show();

  return 0;
}
```

このプログラムからの出力は次のようになります.

```
Enter a string: Hello
Freeing s
Freeing s
Hello
Freeing s
Null pointer assignment
```

samp クラスのデストラクタ関数は，3回呼び出されます．最初は，input()関数が終了し，ローカルオブジェクトのstrがスコープから外れたときです．2回目は，input()関数から返される一時オブジェクトが破棄されるときです．関数からオブジェクトが返されるときには，プログラマにはわかりませんが，戻り値を含む一時オブジェクトが自動的に作成されます．この例の場合，このオブジェクトはstrの単なるコピーであり，これが関数の戻り値となります．したがって，関数が返した後，一時オブジェクトのデストラクタが実行されます．最後に，プログラムの終了時には，main()関数内のobオブジェクトのデストラクタが呼び出されます．

このプログラムの問題点は，デストラクタが最初に実行されるときに，input()関数で文字列入力を格納するために割り当てたメモリが解放されるということです．したがってほかの2つのsampデストラクタ呼び出しで，すでに解放されているメモリを解放しようとする上に，進行中の動的割り当てシステムが破壊されてしまいます．これによって「Null pointer assignment」という実行時メッセージが表示されています(使用するコンパイラ，そのコンパイラのメモリモデルによって，このメッセージが表示される場合と表示されない場合があります)．

重要なのは，関数からオブジェクトを返すときは，返すために一時オブジェクトが作成され，これによってデストラクタ関数が呼び出されるということです．したがって，これによって問題が発生するようなときは，オブジェクトを返さないようにする必要

があります(第5章で説明しますが，コピーコンストラクタを使えばこのような問題を解決できます).

## 練習問題　3.3　関数からのオブジェクトの返し

**1.** whoというクラスを作成し，関数からオブジェクトを返す場合に，オブジェクトが作成されるタイミングと破棄されるタイミングを正確に調べなさい．whoクラスのコンストラクタで，そのオブジェクトを識別するための1文字の引数を受け取ります．コンストラクタでは，オブジェクトの作成時に次のようなメッセージを表示します．

```
Constructing who #x
```

「x」の部分には，各オブジェクトに渡された識別文字を表示します．オブジェクトを破棄する際には，次のようなメッセージを表示します．

```
Destroying who #x
```

この場合も，「x」には識別文字を指定します．最後に，make_who()という名前の関数を作成し，who オブジェクトを返します．

各オブジェクトに一意の名前を付け，このプログラムによって出力される内容を確認しなさい．

**2.** 動的割り当てメモリの不適切な解放のほかに，関数からオブジェクトを返すことが不適切な正しくない状況を考えなさい．

# 3.4　フレンド関数の概要

関数をあるクラスのメンバにすることなく，関数からそのクラスの非公開メンバにアクセスしたいことがあります．この目的のために，C++では**フレンド関数**(friend function)がサポートされています．フレンドは，クラスのメンバではありませんが，そのクラスの非公開メンバにアクセスすることができます．

フレンド関数が役立つ理由は，演算子のオーバーロードと，ある種の入出力関数の作成に関係するものの2つがあります．このようなフレンドの用途については，現段階では説明を控えておきます．フレンド関数を使用する第3の状況としては，1つの関数から，2つ以上の別々のクラスの非公開メンバにアクセスする場合が挙げられます．ここでは，この用途について説明しましょう．

フレンド関数は，メンバではない通常の関数として定義します．ただしフレンドとするクラス宣言の内部に，キーワードfriendを先頭に付けて関数のプロトタイプを含めます．このしくみを理解するために，次の短いプログラムを見てみましょう．

```
// フレンド関数の例
#include <iostream>
using namespace std;

class myclass {
  int n, d;
public:
  myclass(int i, int j) { n = i; d = j; }
  // myclassのフレンドを宣言する
  friend int isfactor(myclass ob);
};

/* ここでフレンド関数を定義する
   この関数は，dがnの因数であれば真を返す
   isfactor()関数の定義では，
   friendキーワードを使用していないことに注意
*/
int isfactor(myclass ob)
{
  if(!(ob.n % ob.d)) return 1;
  else return 0;
}

int main()
{
  myclass ob1(10, 2), ob2(13, 3);

  if(isfactor(ob1)) cout << "2 is a factor of 10¥n";
  else cout << "2 is not a factor of 10¥n";

  if(isfactor(ob2)) cout << "3 is a factor of 13¥n";
  else cout << "3 is not a factor of 13¥n";

  return 0;
}
```

この例では，myclassクラスの宣言の中でコンストラクタ関数と，フレンド関数isfactor()を宣言しています．isfactor()関数はmyclassクラスのフレンドなので，isfactor()関数はmyclassクラスの非公開メンバにアクセスすることができます．このためisfactor()関数では，ob.nとob.dの両方を直接参照することができます．

フレンド関数は，フレンドであるクラスのメンバではないことを理解しておいてください．したがって，オブジェクト名とクラスメンバアクセス演算子(ドット演算子またはアロー演算子)を使用して，フレンド関数を呼び出すことはできません．たとえば，前述の例で次の文を使用することはできません．

```
ob1.isfactor(); // 誤り．isfactor()関数はメンバ関数ではない
```

フレンド関数は，通常の関数と同じ方法で呼び出すことができます．

フレンド関数は，フレンドであるクラスの非公開要素にアクセスすることができますが，そのためにはそのクラスのオブジェクトを介さなければなりません．つまり，myclassクラスのメンバ関数は変数nや変数dに直接アクセスできますが，フレンド関数は，フレンド関数内で宣言されたオブジェクト，または引数として渡されたオブジェクトを使用しないと，これらの変数にアクセスできないのです．

メモ

前の節から重要な点が明らかになっています．メンバ関数は常にそのクラスのオブジェクトに対して実行されるので，メンバ関数から非公開要素を参照する際には直接参照することができます．メンバ関数から非公開要素を参照する際には，メンバ関数の呼び出し時に関数にリンクされているオブジェクトを見て，コンパイラはその非公開要素がどのオブジェクトに属するかを知ることができます．しかしフレンド関数は，どのオブジェクトにもリンクされていません．フレンド関数は単にクラスの非公開要素にアクセスすることを許可されているだけです．したがって，フレンド関数の内部では，特定のオブジェクトを参照せずに非公開メンバを参照しても無意味です．

フレンド関数はクラスのメンバではないので，フレンド関数にはクラスのオブジェクトを1つまたは複数引き渡すのが一般的です．前述のisfactor()関数には，myclassクラスのオブジェクトであるobを渡しています．しかしisfactor()関数は，myclassクラスのフレンド関数なので，obオブジェクトの非公開要素にアクセスすることができます．nとdはmyclassクラスの非公開メンバなので，isfactor()関数がmyclassクラスのフレンド関数でない場合は，ob.dやob.nにアクセスすることはできません．

重要

フレンド関数はメンバではないので，オブジェクト名で修飾することはできません．通常の関数と同じ方法で呼び出す必要があります．

フレンド関数は継承されません．つまり，基本クラスにフレンド関数が含まれていても，その関数は派生クラスのフレンド関数にはなりません．

また，フレンド関数は，複数のクラスのフレンドになることができます．

## 例 3.4　フレンド関数の概要

**1.** フレンド関数の一般的な(そして優れた)用途としては，2つの異なる型のクラスが共通した特性を持っているときに，それを比較することが挙げられます．たとえば，次に示すプログラムでは，car(自動車)というクラスとtruck(トラック)というクラスを作成しています．これらのクラスはどちらも，非公開変数としてそれぞれの乗り物の速度を持っています．

```
#include <iostream>
using namespace std;

class truck; // 前方宣言

class car {
  int passengers;
  int speed;
public:
  car(int p, int s) { passengers = p; speed = s; }
  friend int sp_greater(car c, truck t);
};

class truck {
  int weight;
  int speed;
public:
  truck(int w, int s) { weight = w, speed = s; }
  friend int sp_greater(car c, truck t);
};

/* carの速度がtruckよりも速い場合は，正の値を返す
   速度が同じであれば，0を返す
   truckの速度がcarよりも速い場合は，負の値を返す
*/
int sp_greater(car c, truck t)
{
  return c.speed - t.speed;
}

int main()
{
  int t;
  car c1(6, 55), c2(2, 120);
  truck t1(10000, 55), t2(20000, 72);

  cout << "Comparing c1 and t1:¥n";
  t = sp_greater(c1, t1);
```

```
    if(t<0) cout << "Truck is faster.¥n";
    else if(t==0) cout << "Car and truck speed is the same.¥n";
    else cout << "Car is faster.¥n";

    cout << "¥nComparing c2 and t2:¥n";
    t = sp_greater(c2, t2);
    if(t<0) cout << "Truck is faster.¥n";
    else if(t==0) cout << "Car and truck speed is the same.¥n";
    else cout << "Car is faster.¥n";

    return 0;
  }
```

このプログラムではsp_ greater()という関数を定義しており，これはcarクラスとtruckクラスの両方のフレンド関数です(前述のとおり，1つの関数を複数のクラスのフレンド関数とすることができます)．この関数は，carオブジェクトの速度がtruckオブジェクトよりも速ければ正の値を，両者の速度が同じであれば0を，truckオブジェクトの速度がcarオブジェクトよりも速ければ負の値を返します．

このプログラムでは，C++構文の重要な要素である**前方宣言**(forward declaration)を使用しています(**前方参照**(forward reference)とも呼ばれる)．sp_ greater()関数はcarクラスとtruckクラスの両方を仮引数として受け取りますが，これらのクラスにsp_ greater()関数を含める前に両方を宣言しておくことは，論理的に不可能です．したがって，実際にクラスを宣言することなく，クラスの名前をコンパイラに教える方法が必要となります．このことを前方宣言と呼びます．C++では，識別子がクラス名であることを示すために，クラス名を最初に使う前に次の行を使用します．

```
class class-name;
```

たとえば，前述のプログラムでは，次の行が前方宣言です．

```
class truck;
```

このようにすれば，sp_greater()関数のフレンド宣言でtruckを使用しても，コンパイルエラーは発生しません．

**2.** あるクラスのメンバである関数を，ほかのクラスのフレンド関数として使うことができます．次に示すプログラムは，前述の例を修正し，sp_greater()関数をcarクラスのメンバに，truckクラスのフレンド関数にしたものです．

```
#include <iostream>
using namespace std;
```

```
class truck; // 前方宣言

class car {
  int passengers;
  int speed;
public:
  car(int p, int s) { passengers = p; speed = s; }
  int sp_greater(truck t);
};

class truck {
  int weight;
  int speed;
public:
  truck(int w, int s) { weight = w, speed = s; }
  // スコープ解決演算子を新しい方法で使用している
  friend int car::sp_greater(truck t);
};

/* carの速度がtruckよりも速い場合は，正の値を返す
   速度が同じであれば，0を返す
   truckの速度がcarよりも速い場合は，負の値を返す
*/
int car::sp_greater(truck t)
{
  /* sp_greater()はcarクラスのメンバなので，
     truckオブジェクトだけを渡せばよい */

  return speed-t.speed;
}

int main()
{
  int t;
  car c1(6, 55), c2(2, 120);
  truck t1(10000, 55), t2(20000, 72);

  cout << "Comparing c1 and t1:¥n";
  t = c1.sp_greater(t1);  // carクラスのメンバとして呼び出す
  if(t<0) cout << "Truck is faster.¥n";
  else if(t==0) cout << "Car and truck speed is the same.¥n";
  else cout << "Car is faster.¥n";

  cout << "¥nComparing c2 and t2:¥n";
  t = c2.sp_greater(t2); // carクラスのメンバとして呼び出す
  if(t<0) cout << "Truck is faster.¥n";
  else if(t==0) cout << "Car and truck speed is the same.¥n";
  else cout << "Car is faster.¥n";
```

```
    return 0;
  }
```

truck クラス宣言の中でフレンド関数を宣言する際に，スコープ解決演算子を新しい方法で使っている点に注目してください．この例では，sp_greater()関数がcar クラスのメンバであることをコンパイラに伝えるために，スコープ解決演算子を使用しています．

スコープ演算子の使い方は単純で，クラス名の後にスコープ解決演算子，その後にメンバ名を指定することにより，クラスメンバを完全に特定することができます．

実際にクラスのメンバを参照する際にも，名前を完全に指定しても間違いではありません．しかし，オブジェクトを使ってメンバ関数を呼び出したり，メンバ変数にアクセスしたりする場合に完全名を使うのは冗長であり，めったに使用しません．たとえば，次の例を考えてみましょう．

```
t = c1.sp_greater(t1);
```

これは，冗長なスコープ解決演算子とクラス名(car)を使用して，次のように書くこともできます．

```
t = c1.car::sp_greater(t1);
```

しかし，c1 はcar 型のオブジェクトであり，コンパイラはsp_greater()がcar クラスのメンバであることを知っているので，クラス名を完全に指定する必要はありません．

## 練習問題　3.4　フレンド関数の概要

**1.** 次に示すような，pr1 と pr2 という 2 つのクラスがあるとします．これらのクラスは 1 つのプリンタを共有しています．また，プログラム内で，これらの 2 つのクラスのいずれかのオブジェクトが，オブジェクトを使用しているかどうかを調べる必要があるとします．inuse()という関数を作成し，どちらかのクラスのオブジェクトがプリンタを使用しているときは真を，どちらも使用していないときは偽を返しなさい．この関数は，pr1 と pr2 のフレンド関数として作成します．

```
class pr1 {
  int printing;
  // ...
```

```
public:
  pr1() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
};

class pr2 {
  int printing;
  // ...
public:
  pr2() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
};
```

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** 1つのオブジェクトを，ほかのオブジェクトに代入するために満たさなければならない前提条件を，1つ説明しなさい．

**2.** 次のクラスがあります．

```
class samp {
  double *p;
public:
  samp(double d) {
    p = (double *) malloc(sizeof(double));
    if(!p) exit(1);  // 代入エラー
    *p = d;
  }
  ~samp() { free(p); }
  // ...
};

// ...
samp ob1(123.09), ob2(0.0);
// ...
ob2 = ob1;
```

ob1をob2に代入する際に発生する問題の原因について説明しなさい．

3. 次のクラスがあります.

```
class planet {
  int moons;
  double dist_from_sun; // マイル単位
  double diameter;
  double mass;
public:
  //...
  double get_miles() { return dist_from_sun; }
};
```

light()という関数を作成し，引数としてplanet(惑星)型のオブジェクトを受け取り，太陽からその惑星に光が到達するまでの秒数を返しなさい(光速を毎秒186,000マイルとし，dist_from_sunはマイル単位で指定されているものとします).

4. オブジェクトのアドレスを，引数として関数に渡すことができるかどうか考えなさい.

5. stackクラスを使用して，loadstack()という関数を作成し，アルファベット(aからzまで)を格納したスタックを返しなさい．このスタックを呼び出し元の別のオブジェクトに代入し，そのスタックにアルファベット文字が格納されていることを確認しなさい(アルファベット文字をすべて格納できるように，スタックのサイズを変更するのを忘れないこと).

6. 関数にオブジェクトを渡すとき，またはオブジェクトを関数から返すときに，注意が必要となる理由を説明しなさい.

7. フレンド関数とは何か説明しなさい.

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう.

1. 関数は，仮引数の数または型が異なれば，オーバーロードすることができます.「この章の理解度チェック」問5のloadstack()関数をオーバーロードし，upperという名前の整数を仮引数として受け取るようにしなさい．オーバーロードした関数では，

upperが1である場合は，スタックに大文字のアルファベットを格納します．1ではない場合は，小文字のアルファベットを格納します．

**2.** 3.1節の例3で使用したstrtypeクラスを使用し，strtype型のオブジェクトのポインタを引数として受け取り，そのオブジェクトが指す文字列へのポインタを返すフレンド関数を作成しなさい(つまり，関数からpを返します)．この関数にget_string()という名前を付けます．

**3.** 派生クラスのオブジェクトを，同じクラスから派生したオブジェクトに代入し，基本クラスに関連付けられているデータもコピーされるかどうか調べなさい．このために，次の2つのクラスを使用して，何が起こるかを確認するためのプログラムを作成しなさい．

```
class base {
  int a;
public:
  void load_a(int n) { a = n; }
  int get_a() { return a; }
};

class derived : public base {
  int b;
public:
  void load_b(int n) { b = n; }
  int get_b() { return b; }
};
```

# 4

# 配列，ポインタ，参照

## この章の内容

この章では，オブジェクト配列とオブジェクトポインタを始め，いくつかの重要な事項を説明します．またこの章の最後には，C++の中でも特に重要な機能である参照について説明します．参照は多くのC++機能にとって重要なので，しっかりと理解してください．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. オブジェクトをほかのオブジェクトに代入したときに，何が起こるかを正確に説明しなさい．

2. オブジェクトをほかのオブジェクトに代入したときに発生する問題や副作用について，例を挙げて説明しなさい．

3. 関数に引数としてオブジェクトを渡すと，そのオブジェクトのコピーが作成されます．コピーのオブジェクトに対して，コンストラクタとデストラクタは呼び出されるかどうか説明しなさい．

4. デフォルトでは，関数はオブジェクトに値呼び出しによって渡されます．つまり，関数内でコピーに対して行った処理は，呼び出し時に引数に使用したオブジェクトには影響しません．この原則に対する例外があれば，例を挙げて説明しなさい．

5. 次のクラスを使用して，summation型のオブジェクトを返すmake_sum()という名前の関数を作成しなさい．この関数からユーザーに対して数値を入力するようプロンプトを出し，この値を持つオブジェクトを作成し，そのオブジェクトを呼び出し元プロシージャに返します．また，この関数の動作を確認しなさい．

```
class summation {
  int num;
  long sum; // 数値の総計
public:
  void set_sum(int n);
  void show_sum() {
    cout << num << " summed is " << sum << "¥n";
  }
};
```

```
void summation::set_sum(int n)
{
  int i;
  num = n;
  sum = 0;
  for(i=1; i<=n; i++)
    sum += i;
}
```

**6.** 問5で，set_sum()関数は，summation クラス宣言内でインラインとして宣言されていません．一部のコンパイラでこのようにする必要がある理由を説明しなさい．

**7.** 次のクラスを使用して，myclass型の仮引数を1つ受け取り，numが負数であれば真を，負数でなければ偽を返すisneg()という名前のフレンド関数を追加する方法を示しなさい．

```
class myclass {
  int num;
public:
  myclass(int x) { num = x; }
};
```

**8.** 1つのフレンド関数を，複数のクラスのフレンドとすることができるかどうか述べなさい．

# 4.1　オブジェクトの配列

これまでにも何度か説明してきましたが，オブジェクトは変数であり，ほかの型の変数と同じ機能と属性を持っています．したがって，オブジェクトの配列を作成することも可能です．オブジェクトの配列を宣言する構文は，ほかの型の変数の配列を宣言する際に使用する構文とまったく同じです．さらに，オブジェクト配列へのアクセス方法も，ほかの型の変数配列と同じです．

## 例　4.1　オブジェクトの配列

**1.** 次のプログラムでは，オブジェクトの配列を使用しています．

```
#include <iostream>
using namespace std;

class samp {
```

```
  int a;
public:
  void set_a(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4];
  int i;

  for(i=0; i<4; i++) ob[i].set_a(i);
  for(i=0; i<4; i++) cout << ob[i].get_a( );

  cout << "¥n";

  return 0;
}
```

このプログラムでは，samp型のオブジェクトの要素が4つである配列を作成し，各要素の変数aに0から3までの値を格納しています．各配列要素のメンバ関数を呼び出す方法に注目してください．配列名(この例ではob)に索引を指定し，メンバアクセス演算子を使用して，その後に呼び出すメンバ関数の名前を指定しています．

**2.** クラス型にコンストラクタを含めると，オブジェクトの配列を初期化することができます．次に，ob配列を初期化する例を示します．

```
// 配列を初期化する
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';

  cout << "¥n";
```

```
  return 0;
}
```

このプログラムの出力は次のようになります．

```
-1 -2 -3 -4
```

この例では，-1から-4までの値をobコンストラクタ関数に渡しています．

初期化リストで使用している構文は，次の長い構文を省略したものです．

```
samp ob[4] = { samp(-1), samp(- 2),
               samp(-3), samp(- 4) };
```

しかし，プログラム内で使用している構文の方が一般的に使われています(ただし，この構文は，コンストラクタが引数を1つしか受け取らない配列に対してしか使用できません)．

**3.** オブジェクトの多次元配列を作成することもできます．次のプログラムでは，オブジェクトの2次元配列を作成し，初期化しています．

```
// オブジェクトの2次元配列を作成する
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };

  int i;

  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "¥n";
  }
```

```
  cout << "¥n";

  return 0;
}
```

このプログラムからの出力は，次のようになります．

```
1 2
3 4
5 6
7 8
```

**4.** コンストラクタに複数の引数を渡すこともできます．コンストラクタが複数の引数を受け取るオブジェクト配列を初期化する場合には，前述の長い初期化構文を使用しなければなりません．次に例を示します．

```
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4][2] = {
    samp(1, 2), samp(3, 4),
    samp(5, 6), samp(7, 8),
    samp(9, 10), samp(11, 12),
    samp(13, 14), samp(15, 16)
  };

  int i;

  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][0].get_b() << "¥n";
    cout << ob[i][1].get_a() << ' ';
    cout << ob[i][1].get_b() << "¥n";
  }

  cout << "¥n";

  return 0;
}
```

この例でsampクラスのコンストラクタは，2つの引数を受け取っています．このプログラムでは，sampクラスのコンストラクタを直接呼び出すことによって，main()関数内でob配列を宣言し，初期化しています．C++の正式な構文では，カンマで区切られたリスト内では一度に1つの引数しか受け取ることができないので，この方法が必要となります．リスト内で各エントリに2つ以上の引数を指定することはできません．したがって，複数の引数を受け取るコンストラクタを持つオブジェクト配列を初期化する際には，「省略構文」ではなく「長い構文」を使わなければなりません．

メモ　オブジェクトが1つの引数しか受け取らない場合でも，長い構文はいつでも使用できます．単に，そのような場合は短い構文の方が便利だというだけです．

このプログラムからの出力は次のようになります．

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

## 練習問題　4.1　オブジェクトの配列

1. 次のクラス宣言を使用して，10個の要素を持つ配列を作成し，AからJまでの値を使用して，ch要素を初期化しなさい．また，その配列に本当に値が含まれているかどうか確認しなさい．

```
#include <iostream>
using namespace std;

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};
```

2. 次のクラス宣言を使用して，10個の要素を持つ配列を作成し，1から10までの数値を使用してnumを要素を初期化し，sqr要素をnum要素の2乗に初期化しなさい．

```
#include <iostream>
using namespace std;

class squares {
  int num, sqr;
public:
  squares(int a, int b) { num = a; sqr = b; }
  void show() {cout << num << ' ' << sqr << "¥n"; }
};
```

3. 長い構文を使用して，練習問題1の初期化を修正しなさい(つまり，初期化リストの中で，letters のコンストラクタを明示的に呼び出しなさい).

# 4.2　オブジェクトのポインタ

第2章で説明したとおり，ポインタを使用してオブジェクトにアクセスすることができます．オブジェクトポインタを使用する場合は，ドット(.) 演算子の代わりにアロー(->) 演算子を使用してオブジェクトのメンバにアクセスします．

オブジェクトポインタを使用したポインタ演算は，ほかのデータ型のポインタ演算と同様に，そのオブジェクト型に関して行われます．たとえば，オブジェクトポインタをインクリメントすると，そのポインタは次のオブジェクトを指すようになります．オブジェクトポインタをデクリメントすると，そのポインタは前のオブジェクトを指すようになります．

## 例　4.2　オブジェクトのポインタ

1. 次のプログラムでは，オブジェクトポインタ演算を行っています．

```
// オブジェクトポインタ
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};
```

```
int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;

  samp *p;
  p = ob; // 配列の開始アドレスを取得する

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "¥n";
    p++;  // 次のオブジェクトに進む
  }

  cout << "¥n";

  return 0;
}
```

このプログラムからの出力を次に示します．

```
1 2
3 4
5 6
7 8
```

この出力からもわかるように，p をインクリメントするたびに，p は配列内の次のオブジェクトを指すようになります．

## 練習問題 4.2 オブジェクトのポインタ

**1.** 例 1 を修正し，ob 配列の内容を逆順に表示するようにしなさい．

**2.** 4.1 節の例 3 を修正し，1 つのポインタを使用して 2 次元配列にアクセスするようにしなさい(C と同様，C++ でもすべての配列内の要素はは左から右へ，下から上へ保存されます)．

# 4.3　this ポインタ

C++ には，this という名前の特別なポインタがあります．this はすべてのメンバ変数の呼び出し時に自動的に渡されるポインタで，その呼び出しを行ったオブジェクトを指します．次の文を見てみましょう．

```
ob.f1();  // obはオブジェクト
```

この場合，f1()関数にはobへのポインタが自動的に渡されます．obは関数を呼び出したオブジェクトです．このポインタはthis という名前で参照できます．

thisポインタが渡されるのは，メンバ関数だけです．たとえば，フレンド関数にthisポインタは渡せません．

## 例 4.3　this ポインタ

**1.** メンバ関数から同じクラスのほかのメンバを参照する際には，直接参照することができるので，メンバの前にクラスやオブジェクトを指定する必要はありません．たとえば，次の短いプログラムでは，単純な在庫管理用のクラスを作成しています．

```
// thisポインタの使用例
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
     strcpy(item, i);
     cost = c;
     on_hand = o;
   }
   void show();
};

void inventory::show()
{
  cout << item;
  cout << ": $" << cost;
```

```
  cout << "  On hand: " << on_hand << "¥n";
}

int main()
{
  inventory ob("wrench", 4.95, 4);
  ob.show();
  return 0;
}
```

ご覧のとおり，inventory()コンストラクタ関数とshow()メンバ関数内では，メンバ変数であるitem，cost，on_handを直接参照しています．その理由は，メンバ関数は常に特定のオブジェクトに対して呼び出されるからです．したがって，コンパイラはどのオブジェクトのデータを参照すればよいかを判別することができます．

ただし，もう少し説明しておかなければならないことがあります．メンバ関数を呼び出すと，その関数を呼び出したオブジェクトを指すthisポインタが自動的に渡されます．したがって，前述のプログラムは，次のように書き直すこともできます．

```
// ポインタの使用例
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
     strcpy(this->item, i); // thisポインタを
     this->cost = c;        // 使用してメンバに
     this->on_hand = o;     // アクセスする
   }
   void show();
};

void inventory::show()
{
  cout << this->item; // thisポインタを使用してメンバにアクセスする
  cout << ": $" << this->cost;
  cout << "  On hand: " << this->on_hand << "¥n";
}

int main()
```

```
{
  inventory ob("wrench", 4.95, 4);
  ob.show();
  return 0;
}
```

このプログラムでは，thisポインタを使用してメンバ変数に明示的にアクセスしています．したがって，show()関数内の次の2つの文は同じ意味になります．

```
cost = 123.23;
this->cost = 123.23;
```

実際には，1つ目の構文は2つ目の構文の省略形です．

省略形の方がはるかに単純なので，実際には，このプログラムのようにthisポインタを使用してクラスメンバにアクセスするC++プログラマはいません．しかし，省略形の意味を理解しておくことは重要です．

thisポインタには，演算子のオーバーロード時など，いくつかの用途があります．この用途については第6章で詳しく説明します．現段階では，デフォルトではすべてのメンバ関数に，呼び出し元オブジェクトへのポインタが自動的に渡されるということを理解しておいてください．

## 練習問題　4.3　this ポインタ

**1.** 次のプログラムのクラスメンバへの該当する参照を修正し，thisポインタ参照を明示的に使用しなさい．

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};

void myclass::show()
{
  int t;
```

```
    t = add(); // メンバ関数を呼び出す
    cout << t << "¥n";
  }

  int main()
  {
    myclass ob(10, 14);
    ob.show();
    return 0;
  }
```

# 4.4 new と delete の使用

これまでは，メモリを割り当てるときにはmalloc()関数を使用し，割り当てたメモリを解放するときにはfree()関数を使用してきました．これらは，標準C動的割り当て関数です．C++でもこれらの関数を利用することはできますが，C++ではより安全で便利な方法でメモリを割り当て，解放することができます．つまりC++では，**new**を使用してメモリを割り当て，**delete**を使用してそれを解放することができます．これらの演算子の一般形式を次に示します．

**new 演算子および delete 演算子**

```
p-var = new type;
delete p-var;
```

*type*には，メモリを割り当てるオブジェクトの型指定子を指定します．*p-var*には，その型を指すポインタを指定します．new演算子は，動的に割り当てたメモリ(*type*に指定した型を持つオブジェクトを格納できる十分なサイズを持つ)を指すポインタを返します．delete演算子は，不要になったメモリを解放します．delete 演算子は，new を使用して前に割り当てたポインタに対してしか呼び出すことができません．無効なポインタを使用してdelete演算子を呼び出すと，割り当てシステムが破壊され，プログラムがクラッシュすることがあります．

割り当て要求を満たすだけのメモリがない場合は，2とおりの処置がとられます．new演算子はヌルポインタを返すか，例外を生成します(例外と例外処理については，第11章で説明します．簡単に説明すると，例外とは構造化された形式で管理できる実行時エラーのことです)．標準C++のnew演算子のデフォルトの動作は，割り当て要求を満たすことができない場合に，例外を生成するようになっています．プログラムでこの例外を処理しない場合，プログラムは終了します．厄介なことに，要求を満たすことができない場合にnew演算子がとる正確な処置は，過去数年の間に数回にわたって変更されていま

す．したがって，使用するコンパイラによっては，標準C++で定義されているnew演算子の動作が実装されていないこともあります．

C++が最初に開発されたとき，new演算子は割り当てに失敗するとヌルを返していました．これが後に修正されて，new演算子は失敗したときに例外を生成するようになりました．最終的に，new演算子が失敗したときには，デフォルトで例外を生成することが決定されましたが，オプションとしてヌルポインタを返すこともできます．つまり，コンパイラの製造元や時期によって，new演算子は異なる方法で実装されたのです．たとえば，本書の執筆時点では，MicrosoftのVisual C++はnew演算子が失敗したときにヌルポインタを返します．Borland C++は例外を生成します．最終的にはすべてのコンパイラが標準C++に合わせてnew演算子を実装するものと思われますが，現在のところ，失敗したときのnew演算子の動作を正確に知るには，コンパイラのマニュアルを調べるしかありません．

new演算子が割り当ての失敗を示す方法は2とおりあり，またどちらの動作を取るかはコンパイラによって異なるので，本書のプログラムコードは，両方の状況に対処できるように作成します．本書で紹介するすべてのプログラムコードでは，new演算子から返されたポインタがヌルかどうかを調べます．これによって，失敗したときにヌルを返すnew演算子を実装するコンパイラに対処しますが，このことで例外を生成するコンパイラに害が及ぶことはありません．new演算子が失敗したときに例外を生成するコンパイラを使用している場合，プログラムは単に終了します．後で例外処理について説明する際に，new演算子について再び取り上げ，より優れた方法で割り当てエラーに対処する方法を学びます．また，エラーが発生したときに常にヌルポインタを返す形式のnew演算子についても説明します．

ただし，本書で紹介するプログラムでは，1つのプログラムで割り当てているバイト数はほんの少量なので，new演算子が失敗することはないでしょう．

new演算子とdelete演算子の機能はmalloc()関数とfree()関数に似ていますが，new演算子とdelete演算子にはいくつかの利点があります．第1に，new演算子は指定の型のオブジェクトを格納するのに十分なメモリを自動的に割り当てます．必要なバイト数を調べるために，sizeofなどを使用する必要はありません．これによって，エラーが発生する可能性が減ります．第2に，new演算子は指定した型のポインタを自動的に返します．malloc()関数を使用してメモリを割り当てたときのように，明示的に型キャストを行う必要はありません(次の「メモ」を参照)．第3に，new演算子とdelete演算子はどちらもオーバーロードできるので，独自の割り当てシステムを簡単に作成することができます．第4に，動的に割り当てたオブジェクトを初期化することができます．最後に，プログラムに<cstdlib>をインクルードする必要がありません．

メモ

Cでは，malloc()関数の戻り値をポインタに代入するときに，型のキャストを行う必要はありません．これは，malloc()関数から返されるvoid *は，代入文の左辺にあるポインタの型と互換性のあるポインタに自動的に変換されるからです．しかし，C++ではこのことが当てはまりません．C++でmalloc()関数を使うときには，明示的に型を変換する必要があります．CとC++にこのような違いがある理由は，C++では型のチェックが厳密だからです．

new演算子とdelete演算子について学んだので，malloc()関数とfree()関数の代わりにこれらの演算子を使用してみましょう．

## 例　4.4　new と delete の使用

**1.** 次のプログラムでは，整数を格納するメモリを割り当てています．

```
// new演算子とdelete演算子を使用する単純な例
#include <iostream>
using namespace std;

int main()
{
  int *p;

  p = new int; // 整数のメモリを割り当てる
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  *p = 1000;
  cout << "Here is integer at p: " << *p << "¥n";
  delete p;  // メモリを解放する
  return 0;
}
```

new演算子によって返される値を，使用前に調べている点に注目してください．前述のとおり，この検査は，new演算子が失敗したときにヌルを返すように実装されているコンパイラでしか意味がありません．

**2.** 次のプログラムでは，オブジェクトを動的に割り当てています．

```
// 動的オブジェクトを割り当てる
#include <iostream>
```

```
using namespace std;

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;

  p = new samp; // オブジェクトを割り当てる
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  p->set_ij(4, 5);
  cout << "Product is: " << p->get_product() << "¥n";
  return 0;
}
```

**練習問題　4.4　new と delete の使用**

1. new 演算子を使用して float 型，long 型，char 型に動的にメモリを割り当てなさい．これらの動的変数に値を格納し，その値を表示しなさい．最後に，delete演算子を使用して，動的に割り当てたメモリをすべて解放しなさい．

2. 名前と電話番号を保存するクラスを作成しなさい．new演算子を使用してこのクラスのオブジェクトを動的に割り当て，このオブジェクト内のこれらのフィールドに自分の名前と電話番号を格納しなさい．

3. new 演算子が割り当てに失敗したときにとる処置を２つ説明しなさい．

# 4.5　new と delete の詳細

この節では，new 演算子と delete 演算子のほかの機能を２つ説明します．１つ目は，動的に割り当てたオブジェクトに初期値を与える機能です．２つ目は，配列を動的に割り当てる機能です．

動的に割り当てたオブジェクトに初期値を与えるには，次の構文で new 文を使用します．

new 文

```
p-var = new type (initial-value);
```

1 次元配列を動的に割り当てるには，次の構文で new 文を使用します．

new 文

```
p-var = new type [size];
```

この文を実行すると，*p-var* は *size* 個の要素を持つ *type* 型の配列の先頭を指すようになります．さまざまな技術的理由から，動的に割り当てた配列を初期化することはできません．

動的に割り当てた配列を削除するには，次の構文で delete 文を使用します．

delete 文

```
delete [ ]  p-var;
```

この構文を使用すると，コンパイラは配列内の各要素ごとにデストラクタ関数を呼び出します．ただし，*p-var* が複数回解放されるわけではありません．*p-var* は 1 回しか解放されません．

メモ

古いコンパイラを使用する場合は，削除する配列のサイズを大括弧([ ])で囲み，delete文に指定しなければならないこともあります．これは，初期のC++定義で必須とされていました．ただし，最近のコンパイラではサイズを指定する必要はありません．

## 例 4.5　new と delete の詳細

**1.**　次のプログラムでは，整数に動的にメモリを割り当て，そのメモリを初期化しています．

```
// 動的変数を初期化する例
#include <iostream>
using namespace std;

int main()
{
  int *p;

  p = new int(9); // 初期値として9を格納する
```

```
    if(!p) {
      cout << "Allocation error¥n";
      return 1;
    }

    cout << "Here is integer at p: " << *p << "¥n";

    delete p;  // メモリを解放する

    return 0;
  }
```

このプログラムを実行すると，9が表示されます．これはpが指すメモリに格納した初期値です．

**2.** 次のプログラムでは，動的に割り合てたオブジェクトに初期値を渡しています．

```
// 動的オブジェクトを割り当てる
#include <iostream>
using namespace std;

class samp {
  int i, j;
public:
  samp(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;

  p = new samp(6, 5); // オブジェクトを割り当てて初期化する

  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  cout << "Product is: " << p->get_product() << "¥n";

  delete p;

  return 0;
}
```

sampオブジェクトを割り当てると，コンストラクタが自動的に呼び出されて，値6と5が渡されます．

**3.** 次のプログラムでは，整数の配列を割り当てます．

```
// new演算子とdelete演算子の単純な例
#include <iostream>
using namespace std;

int main()
{
  int *p;

  p = new int [5]; // 5つの整数用のメモリを割り当てる

  // 割り当てが成功したことを常に確認する
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  int i;

  for(i=0; i<5; i++) p[i] = i;

  for(i=0; i<5; i++) {
    cout << "Here is integer at p[" << i << "]: ";
    cout << p[i] << "¥n";
  }

  delete [] p;  // メモリを解放する

  return 0;
}
```

このプログラムの出力は次のようになります．

```
Here is integer at p[0]: 0
Here is integer at p[1]: 1
Here is integer at p[2]: 2
Here is integer at p[3]: 3
Here is integer at p[4]: 4
```

**4.** 次のプログラムでは，オブジェクトの動的配列を作成しています．

```
// 動的オブジェクトを割り当てる
#include <iostream>
using namespace std;

class samp {
  int i, j;
```

```
public:
  void set_ij(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;
  int i;

  p = new samp [10]; // オブジェクト配列を割り当てる
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Product [" << i << "] is: ";
    cout << p[i].get_product() << "¥n";
  }

  delete [] p;
  return 0;
}
```

このプログラムの出力は次のようになります．

```
Product [0] is: 0
Product [1] is: 1
Product [2] is: 4
Product [3] is: 9
Product [4] is: 16
Product [5] is: 25
Product [6] is: 36
Product [7] is: 49
Product [8] is: 64
Product [9] is: 81
```

**5.** 次のプログラムは，前の例に追加して，samp クラスにデストラクタを用意しています．p を解放すると，各要素のデストラクタが呼び出されます．

```
 // 動的オブジェクトを割り当てる
#include <iostream>
using namespace std;
```

```
class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  ~samp() { cout << "Destroying...¥n"; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;
  int i;

  p = new samp [10]; // オブジェクト配列を割り当てる
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Product [" << i << "] is: ";
    cout << p[i].get_product() << "¥n";
  }

  delete [] p;
  return 0;
}
```

このプログラムの出力は次のようになります.

```
Product [0] is: 0
Product [1] is: 1
Product [2] is: 4
Product [3] is: 9
Product [4] is: 16
Product [5] is: 25
Product [6] is: 36
Product [7] is: 49
Product [8] is: 64
Product [9] is: 81
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
```

```
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
```

ご覧のとおり，samp クラスのデストラクタは，配列内の要素ごとに1回ずつ，全部で10回呼び出されています．

### 練習問題　4.5　new と delete の詳細

**1.** new 演算子を使用して次のプログラムコードを書き換えなさい．

```
char *p;

p = (char *) malloc(100);
// ...
strcpy(p, "This is a test");
```

ヒント　文字列とは単なる文字の配列のことです．

**2.** new 演算子を使用して，double 型の変数を割り当て，初期値として -123.0987 を格納する方法を説明しなさい．

## 4.6　参照

C++には，ポインタに関連する機能として，参照が用意されています．**参照**(reference)とは，あらゆる点で変数の別名として動作する暗黙的なポインタ(implicit pointer)のことです．

参照の用途は3とおりあります．第1に，参照を関数に渡すことができます．第2に，関数から参照を返すことができます．第3に，独立した参照を作成することができます．次に，これらの各用途について説明します．まずは参照を仮引数として渡す方法を見てみましょう．参照の最も重要な用途は，関数への仮引数として使うことです．**参照仮引数**とは何か，どのように動作するかを理解するために，仮引数としてポインタ(参照ではなく)を使用するプログラムを先に見てみましょう．

```
#include <iostream>
using namespace std;

void f(int *n);   // ポインタ仮引数を使用する
```

```
int main()
{
  int i = 0;

  f(&i);

  cout << "Here is i's new value: " << i << '¥n';

  return 0;
}

void f(int *n)
{
  *n = 100; // nが指す引数に100を格納する
}
```

このプログラムのf()関数では，nが指す整数に100を格納しています．このプログラムでは，main()関数の中でf()関数にiのアドレスを渡しています．したがって，f()関数が終了した後，iには100が格納されています．

このプログラムは，仮引数としてポインタを使用して，参照呼び出し仮引数機構を手作業で作成しています．Cプログラムでは，これが参照呼び出しを行う唯一の方法です．しかし，C++では参照仮引数を使用して，このプロセスを完全に自動化することができます．その方法を学ぶために，前述のプログラムを修正しましょう．次に，参照仮引数を使用するプログラムを示します．

```
#include <iostream>
using namespace std;

void f(int &n); // 参照仮引数を宣言する

int main()
{
  int i = 0;

  f(i);

  cout << "Here is i's new value: " << i << '¥n';

  return 0;
}

// f()関数は参照仮引数を使用する
void f(int &n)
{
  // 次の文では*が必要ない
  n = 100; // f()関数を呼び出すのに使用した引数に100を格納する
}
```

このプログラムについて詳しく説明しましょう．まず，参照変数(参照仮引数)を宣言するには，変数名の前に&を付けます．これで，変数nをf()関数への仮引数として宣言することができます．nは参照なので*演算子を使用する必要はありません．代わりに，f()関数内でnを使用するたびに，nはf()関数を呼び出すのに使われた引数へのポインタとして扱われます．つまり，次の文では，f()関数を呼び出す際に使用した変数(この例ではi)に，100を格納しているということです．

```
n = 100;
```

さらに，f()関数を呼び出す際には，引数の前に&を付ける必要がありません．その代わり，f()関数は参照仮引数を受け取るように宣言されているので，f()関数には自動的に引数のアドレスが渡されます．

要点をまとめると，参照仮引数を使用する場合，コンパイラは，引数として使われた変数のアドレスを自動的に渡します．変数の前に&を付けて，手作業で引数のアドレスを生成する必要はありません(実際には，変数の前に&を付けてはいけません)．さらに関数内では，コンパイラは参照仮引数が指す変数を自動的に使用します．*を使用する必要はありません(使用してはいけません)．つまり参照仮引数では，参照呼び出し機構が完全に自動化されているのです．

参照が指すものを変更することはできません．たとえば，次の文を前述のプログラムのf()関数内で使用したとします．

```
n++;
```

この文を使用しても，nの参照先はmain()関数内のi以外のものに変わることはありません．この文では，nをインクリメントするのではなく，参照されている変数(この場合はi)の値をインクリメントしていることになります．

参照仮引数は，ポインタ仮引数と比べて多少なりとも優れている点がいくつかあります．第1に，実用的な面では，引数のアドレスを渡すことに気をつかう必要がありません．参照仮引数を使用するときは，自動的にアドレスが渡されます．第2に，多くのプログラマは明示的なポインタを使う方法は扱いにくく，参照仮引数の方がわかりやすく優れた方法であると考えています．第3に，次の節で説明しますが，オブジェクトを参照として関数に渡す際には，コピーが作成されません．これは，引数のコピーのデストラクタ関数が呼び出されることによって，プログラム内のほかの場所で必要となるデータが破壊されてしまうという問題を避ける1つの方法です．

## 例 4.6 参照

**1.** 引数を参照呼び出しする関数の典型的な例として，渡された2つの引数の値を交換する関数があります．次に，参照を使用して2つの整数引数を入れ替えるswapargs()関数を示します．

```
#include <iostream>
using namespace std;

void swapargs(int &x, int &y);

int main()
{
  int i, j;

  i = 10;
  j = 19;

  cout << "i: " << i << ", ";
  cout << "j: " << j << "¥n";

  swapargs(i, j);

  cout << "After swapping: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "¥n";

  return 0;
}

void swapargs(int &x, int &y)
{
  int t;

  t = x;
  x = y;
  y = t;
}
```

参照ではなくポインタを使用してswapargs()関数を作成すると，次のようになります．

```
void swapargs(int *x, int *y)
{
  int t;

  t = *x;
  *x = *y;
```

```
  *y = t;
}
```

このように，参照を使用してswapargs()関数を作成した場合には，＊演算子を挿入する必要がなくなります．

**2.** 次のプログラムでは，round()関数を使用してdouble型の値を丸めています．丸める値を参照によって渡します．

```
#include <iostream>
#include <cmath>
using namespace std;

void round(double &num);

int main()
{
  double i = 100.4;

  cout << i << " rounded is ";
  round(i);
  cout << i << "¥n";

  i = 10.9;
  cout << i << " rounded is ";
  round(i);
  cout << i << "¥n";

  return 0;
}

void round(double &num)
{
  double frac;
  double val;

  // numを整数部と小数部に分解する
  frac = modf(num, &val);

  if(frac < 0.5) num = val;
  else num = val+1.0;
}
```

round()関数では，modf()という比較的わかりにくい標準ライブラリ関数を使用して，数値を整数部と小数部に分解しています．modf()関数の戻り値は小数部で，整数部はmodf()関数の２つ目の仮引数が指す変数に格納されます．

**練習問題　4.6　参照**

1. 整数仮引数の符号を逆にするneg()という関数を作成しなさい．この関数を2とおりの方法で作成し，1つはポインタ仮引数を使用し，もう1つは参照仮引数を使用します．また，短いプログラムを作成して両者の動作を確認しなさい．

2. 次のプログラムの誤りを指摘しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
using namespace std;

void triple(double &num);

int main()
{
  double d = 7.0;
  triple(&d);
  cout << d;
  return 0;
}

// num値を3倍する
void triple(double &num)
{
  num = 3 * num;
}
```

3. 参照仮引数の利点について説明しなさい．

# 4.7　オブジェクト参照の引き渡し

第3章で説明したとおり，デフォルトの値呼び出し機構を使用して関数にオブジェクトを渡した場合は，そのオブジェクトのコピーが作成されます．仮引数のコンストラクタ関数は呼び出されませんが，関数の終了時にはデストラクタ関数が呼び出されます．以前にも説明しましたが，場合によってはこれが深刻な問題となることがあります．たとえば，デストラクタで動的メモリを解放するような場合です．

この問題に対する解決法の1つとして，オブジェクトを参照によって渡す方法があります(このほかの解決法として，コピーコンストラクタを使用する方法もありますが，これについては第5章で説明します)．オブジェクトを参照によって渡すと，コピーが作成されないので，関数の終了時にデストラク

タ関数が作成されることもありません．ただし，この場合は関数内でオブジェクトに加えた変更が，引数として使用したオブジェクトにも影響するので注意が必要です．

メモ　参照はポインタではないことに注意してください．したがって，参照によってオブジェクトを渡したときは，メンバアクセス演算子にはアロー(->)ではなくドット(.)を使用します．

## 例 4.7 オブジェクト参照の引き渡し

**1.** ここでは，オブジェクトを参照によって渡す方法の便利さを示すために，2つのプログラムを紹介します．次の1つ目のプログラムは，f()関数にmyclassオブジェクトを値によって渡しています．

```
#include <iostream>
using namespace std;

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Constructing " << who << "¥n";
  }
  ~myclass() { cout << "Destructing " << who << "¥n"; }
  int id() { return who; }
};

// oを値によって渡す
void f(myclass o)
{
  cout << "Received " << o.id() << "¥n";
}

int main()
{
  myclass  x(1);
  f(x);
  return 0;
}
```

この関数からの出力は次のようになります．

```
Constructing 1
Received 1
```

```
Destructing 1
Destructing 1
```

ご覧のとおり，デストラクタ関数が2回呼び出されています．1回目はf()関数の終了時で，オブジェクト1のコピーが破棄されたときです．2回目はプログラムの終了時です．

f()関数で参照仮引数を使用するようにこのプログラムを修正すると，コピーは作成されません．したがって，f()関数の終了時にもデストラクタは呼び出されません．

```
#include <iostream>
using namespace std;

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Constructing " << who << "¥n";
  }
  ~myclass() { cout << "Destructing " << who << "¥n"; }
  int id() { return who; }
};

// oを参照によって渡す
void f(myclass &o)
{
  // .演算子を使用していることに注意
  cout << "Received " << o.id() << "¥n";
}

int main()
{
  myclass  x(1);
  f(x);
  return 0;
}
```

このプログラムからの出力は次のようになります．

```
Constructing 1
Received 1
Destructing 1
```

**重要**　参照を使用してオブジェクトのメンバにアクセスするときは，アロー演算子ではなくドット演算子を使用します．

## 練習問題 4.7 オブジェクト参照の引き渡し

**1.** 次のプログラムの誤りを指摘しなさい．また，参照仮引数を使用してこのプログラムを修正しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;
  p = new char [l];

  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, s);
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "¥n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
```

```
    show(b);

    return 0;
  }
```

# 4.8　参照の返し

関数から参照を返すことができます．第6章でも説明しますが，ある種の演算子をオーバーロードする際には，参照を返すと大変役に立ちます．このほかに，参照を返すことによって，代入文の左辺で関数を使用することもできます．この効果は，驚くほど強力です．

## 例 4.8　参照の返し

1. 手始めとして，参照を返す関数を含むごく単純なプログラムを示します．

```
// 参照を返す関数の例
#include <iostream>
using namespace std;

int &f(); // 参照を返す
int x;

int main()
{
  f() = 100; // f()関数によって返される参照に100を代入する
  cout << x << "¥n";
  return 0;
}

// int型の参照を返す
int &f()
{
  return x;  // xへの参照を返す
}
```

このプログラムのf()関数は，整数への参照を返すように宣言されています．関数の本体では，次の文を使用しています．

```
return x;
```

この文は，グローバル変数xの値を返すのではなく，xのアドレス(参照という形で)を自動的に返します．f()関数がxへの参照を返すので，main()関数内の次の文では，xに100を代入していることになります．

```
f() = 100;
```

まとめると，f()関数は参照を返します．したがって，f()関数を代入文の左辺で使用すると，この関数の戻り値である参照に対して代入が行われます．f()関数はxへの参照を返すので(この例では)，xが100という値を受け取ります．

**2.** 参照を返す際には，参照先のオブジェクトがスコープから外れないように注意する必要があります．例として，次のプログラムを考えてみましょう．これはf()関数を少し修正したものです．

```
// int型の参照を返す
int &f()
{
  int x;      // xはローカル変数
  return x;   // xへの参照を返す
}
```

この場合，変数xはf()関数に対してローカルなので，f()関数が終了するとスコープから外れます．つまり，f()関数から返される参照は役に立ちません．

メモ　一部のC++コンパイラでは，ローカル変数への参照を返すことができません．しかし，この種の問題は，オブジェクトを動的に割り当てるときなど，別の形で明らかになることもあります．

**3.** 参照を返す優れた用途としては，境界付き配列型を作成することが挙げられます．CとC++では，配列の境界チェックは行われません．したがって，配列のオーバーフローやアンダーフローが起こることがあります．しかし，C++では自動境界チェックを行う配列クラスを作成することができます．配列クラスには2つの主要関数を含めます．1つは情報を配列に格納するための関数，もう1つは情報を取得するための関数です．これらの関数を使用して，配列境界を越えていないかどうかを実行時にチェックすることができます．

次のプログラムでは，文字用の境界チェック配列を作成しています．

```
// 単純な境界配列の例
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

class array {
  int size;
  char *p;
public:
  array(int num);
  ~array() { delete [] p; }
  char &put(int i);
  char get(int i);
};

array::array(int num)
{
  p = new char [num];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  size = num;
}

// 配列に情報を格納する
char &array::put(int i)
{
  if(i<0 || i>=size) {
    cout << "Bounds error!!!¥n";
    exit(1);
  }
  return p[i];  // p[i]への参照を返す
}

// Get something from the array.
char array::get(int i)
{
  if(i<0 || i>=size) {
    cout << "Bounds error!!!¥n";
    exit(1);
  }
  return p[i]; // 文字を返す
}

int main()
{
  array a(10);

  a.put(3) = 'X';
  a.put(2) = 'R';
```

```
    cout << a.get(3) << a.get(2);
    cout << "¥n";

    // 実行時境界エラーを生成する
    a.put(11) = '!';

    return 0;
}
```

このプログラムは，参照を返す非常に実用的な関数の例なので，詳しく説明することにしましょう．put()関数は，仮引数iによって指定された配列要素への参照を返します．この参照を代入文の左辺で使用すると，iに指定した索引が境界の範囲内にあれば，配列に情報を格納することができます．get()関数では逆の処理を行っており，索引が範囲内にあれば，指定された索引位置に保存されている値を返します．このようにして配列を管理することを，**安全配列**(safe array)と呼びます(第6章では，安全配列を作成するより優れた方法を紹介します)．

また，このプログラムでは，new演算子を使用して配列を動的に割り当てている点に注目してください．これによって，さまざまな長さの配列を宣言することができます．

前述のとおり，このプログラムで行っている境界チェックは，C++では非常に実用的です．実行時に配列境界を調べる必要がある場合は，この方法を用いるとよいでしょう．ただし，境界チェックを行うと，配列へのアクセス速度が落ちることを覚えておいてください．したがって，境界チェックは，本当に配列境界を越える可能性がある場合にのみ行うようにしましょう．

## 練習問題　4.8　参照の返し

**1.** 整数を格納する2行3列の2次元の安全な配列を作成しなさい．また，この配列の動作を確認しなさい．

**2.** 次のプログラムコードが正しいかどうか考えなさい．正しくない場合は，その理由を説明しなさい．

```
int &f();
.
.
.
int *x;
x = f();
```

# 4.9　独立参照と制限

あまり一般的ではありませんが，C++では**独立参照**(independent reference)という参照を使用することもできます．独立参照とは，単にほかの変数の別名である参照変数のことです．参照には新しい値を代入することができないので，独立参照は宣言時に初期化する必要があります．

メモ　独立参照はときどき使われるので，知っておくことは重要です．しかし，ほとんどのプログラマは独立参照を使う必要はなく，使用するとプログラムがわかりにくくなると考えています．さらに，C++で独立参照が用意されている主な理由は，あえて削除するべき理由がなかったということです．ほとんどの場合は，独立参照を使用しない方がよいでしょう．

すべての種類の参照には，数多くの制限があります．ほかの参照を参照することはできません．参照のアドレスを取得することはできません．参照の配列を作成することはできず，ビットフィールドを参照することもできません．参照は，クラスのメンバ，戻り値，関数仮引数のいずれかでない場合は，初期化しなければなりません．

重要　参照はポインタと似ていますが，ポインタではありません．

## 例　4.9　独立参照と制限

1.　次のプログラムでは，独立参照を使用しています．

```
#include <iostream>
using namespace std;
int main()
{
  int x;
  int &ref = x; // 独立参照を作成する

  x = 10;         // この2つの文の
  ref = 10;       // 機能は同等である

  ref = 100;
  // 100を2回出力する
  cout << x << ' ' << ref << "¥n";

  return 0;
}
```

このプログラムでは，独立参照refをxの別名として使用しています．実際には，xとrefは同じものであると考えることができます．

**2.** 独立参照を使用して，定数を参照することができます．たとえば，次の文は有効です．

```
const int &ref = 10;
```

繰り返しますが，この種の参照を使用する利点はほとんどありません．しかし，ほかのプログラムで使われているのを見かけることはあるでしょう．

### 練習問題 4.9 独立参照と制限

**1.** 独立参照の用途を自分で考えてみましょう．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** 次のクラスを使用して，2行5列の2次元配列を作成し，配列内の各オブジェクトに任意の初期値を設定しなさい．次に，配列の内容を表示しなさい．

```
class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << "¥n"; }
};
```

**2.** 前問で作成したプログラムを修正し，ポインタを使用して配列にアクセスしなさい．

**3.** thisポインタとは何か説明しなさい．

**4.** new演算子とdelete演算子の一般形式を説明しなさい．また，malloc()関数とfree()関数の代わりにこれらの演算子を使う利点を説明しなさい．

**5.** 参照とは何か説明しなさい．また，参照仮引数を使用する利点を1つ挙げなさい．

**6.** double 型の参照仮引数を1つ受け取る recip() という名前の関数を作成しなさい．この関数で仮引数の値を逆数に変更します．また，この関数の動作を確認するプログラムを作成しなさい．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** オブジェクトを指すポインタがある場合，オブジェクトのメンバにアクセスするにはどの演算子を使用しますか．

**2.** 第2章では，文字列用のメモリを動的に割り当てる strtype クラスを作成しました．new 演算子と delete 演算子を使用して，次に示す strtype クラスを修正しなさい．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
```

```
  cout << "Freeing p¥n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  return 0;
}
```

**3.** 前の章で作成した中から任意のプログラムを選び，参照を使用するように修正しなさい．

# 5

# 関数オーバーロード

## この章の内容

この章では，関数オーバーロードについてさらに詳しく学びます．関数オーバーロードについては本書の最初の部分でも説明しましたが，まだ説明しなければならないことがいくつかあります．この章では，コンストラクタ関数をオーバーロードする方法や，コピーコンストラクタを作成する方法，関数のデフォルト引数を定義する方法，オーバーロードのあいまいさを防ぐ方法について説明します．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. 参照とは何か説明しなさい．また，参照の重要な用途を2つ説明しなさい．

2. new演算子を使用して，float型とint型を割り当てなさい．また，delete演算子を使用して，これらの変数を解放しなさい．

3. 動的変数の初期化に使用するnew演算子の一般形式を説明しなさい．また，具体的な使用例を示しなさい．

4. 次のクラスを使用して，10個の要素を持つ配列を初期化し，xに1から10までの値を格納しなさい．

```
class samp {
  int x;
public:
  samp(int n) { x = n; }
  int getx() { return x; }
};
```

5. 参照仮引数の利点と欠点を1つずつ挙げなさい．

6. 動的に割り当てた配列を初期化することができるかどうか述べなさい．

7. 次のプロトタイプを使用して，orderで指定された桁数までnumの桁数を上げるmag()という関数を作成しなさい．

```
void mag(long &num, long order);
```

たとえば，numが4，orderが2の場合，mag()関数が終了したときに，numが400になるようにします．また，この関数の動作を確認するプログラムを作成しなさい．

# 5.1　コンストラクタ関数のオーバーロード

クラスのコンストラクタ関数は，オーバーロードすることができます(デストラクタ関数は，オーバーロードできません)．コンストラクタ関数をオーバーロードする目的は，主に3つあります．1つ目は柔軟性を得ること，2つ目は配列をサポートすること，3つ目はコピーコンストラクタを作成することです．この節では，最初の2つの理由について説明します．コピーコンストラクタについては，次の節で説明します．

サンプルプログラムを見る際には，クラスのオブジェクトを作成する場合，その作成方法ごとに対応するコンストラクタ関数がなければならないことに注意してください．対応するコンストラクタを持たないオブジェクトを作成しようとすると，コンパイル時にエラーが発生します．これが，C++プログラムでコンストラクタ関数のオーバーロードがよく用いられる理由です．

## 例　5.1　コンストラクタ関数のオーバーロード

**1.** コンストラクタ関数のオーバーロードの用途として最も一般的なのは，オブジェクトを初期化するかしないかを選択できるようにすることです．次のプログラムでは，o1は初期化していますが，o2は初期化していません．

空の引数リストを持つコンストラクタを削除すると，初期化しないsamp型オブジェクトに対応するコンストラクタが存在しなくなるので，このプログラムはコンパイルできなくなります．その逆も同様です．仮引数付きのコンストラクタを削除すると，初期化するsamp型オブジェクトに対応するコンストラクタが存在しなくなるので，プログラムをコンパイルできません．このプログラムを正しくコンパイルするには，どちらのコンストラクタも必要です．

```
#include <iostream>
using namespace std;

class myclass {
  int x;
public:
  // コンストラクタを2とおりにオーバーロードする
  myclass() { x = 0; }        // 初期化値を指定しない
  myclass(int n) { x = n; } // 初期化値を指定する
  int getx() { return x; }
};

int main()
```

```
{
  myclass o1(10); // 初期値を指定して宣言する
  myclass o2;     // 初期値を設定せずに宣言する

  cout << "o1: " << o1.getx() << '¥n';
  cout << "o2: " << o2.getx() << '¥n';

  return 0;
}
```

**2.** このほかに，コンストラクタ関数をオーバーロードする主な理由としては，個々のオブジェクトとオブジェクトの配列をプログラム内で同時に使用できるようにすることがあります．過去のプログラミング経験からすでにご存じでしょうが，1つの変数を初期化することはよくありますが，配列を初期化するのはあまり一般的ではありません(配列には，プログラムの実行中にしかわからない情報を使用して値を格納するのが一般的です)．したがって，初期化しないオブジェクト配列と初期化するオブジェクト配列を両方ともサポートするには，初期化を行うコンストラクタと行わないコンストラクタを用意する必要があります．

たとえば，例1のmyclassクラスを使用する場合，次の宣言はどちらも有効です．

```
myclass ob(10);
myclass ob[5];
```

仮引数付きのコンストラクタと仮引数なしのコンストラクタを両方とも用意することによって，プログラムでは必要に応じて初期化するオブジェクトと初期化しないオブジェクトをどちらでも作成することができます．

当然ながら，仮引数付きと仮引数なしのコンストラクタを両方とも定義したら，それらを使用して初期化する配列と初期化しない配列を作成することができます．たとえば，次のプログラムではmyclass型の2つの配列を宣言しています．1つは初期化し，もう1つは初期化していません．

```
#include <iostream>
using namespace std;

class myclass {
  int x;
public:
  // コンストラクタを2とおりにオーバーロードする
  myclass() { x = 0; }        // 初期化値を指定しない
  myclass(int n) { x = n; } // 初期化値を指定する
  int getx() { return x; }
```

```
};

int main()
{
  // 初期値を設定せずに宣言する
  myclass o1[10];

  // 初期値を設定して宣言する
  myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int i;

  for(i=0; i<10; i++) {
    cout << "o1[" << i << "]: " << o1[i].getx();
    cout << '¥n';
    cout << "o2[" << i << "]: " << o2[i].getx();
    cout << '¥n';
  }

  return 0;
}
```

この例では，コンストラクタ関数を使用して，o1のすべての要素を0に設定しています．o2の要素は，プログラム内に示している値を使用して初期化しています．

**3.** コンストラクタ関数をオーバーロードすることによって，プログラマは最も便利なオブジェクト初期化方法を選択することができます．この方法を確認するために，次のサンプルプログラムでは日付を保持するクラスを作成しています．このプログラムではdate()コンストラクタを2とおりにオーバーロードします．1つのコンストラクタでは日付を文字列として受け取り，もう1つのコンストラクタでは日付を3つの整数として受け取ります．

```
#include <iostream>
#include <cstdio> // sscanf()関数用にインクルードする
using namespace std;

class date {
  int day, month, year;
public:
  date(char *str);
  date (int m, int d, int y) {
    day = d;
    month = m;
    year = y;
  }
  void show() {
```

```
    cout << month << '/' << day << '/';
    cout << year << '¥n';
  }
};

date::date(char *str)
{
  sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

int main()
{
  // 文字列を使用してdateオブジェクトを作成する
  date sdate("12/31/99");

  // 整数を使用してdateオブジェクトを作成する
  date idate(12, 31, 99);

  sdate.show();
  idate.show();
  return 0;
}
```

このプログラムのようにdate()コンストラクタをオーバーロードすると，使用するときの状況に最適なコンストラクタを自由に使用できるようになります．たとえば，ユーザー入力を基にdateオブジェクトを作成する場合は，文字列を受け取るコンストラクタを使用するのが便利でしょう．しかし，何らかの内部処理を通してdateオブジェクトを作成する場合には，3つの整数を受け取るコンストラクタの方が使いやすいはずです．

コンストラクタ関数は必要なだけ何とおりにもオーバーロードすることができますが，あまり数多くオーバーロードすると逆効果です．スタイルという観点から言えば，頻繁に発生しそうな状況のみに合わせてコンストラクタをオーバーロードするのが最良の方法です．たとえば，date()関数を3とおりにオーバーロードして，ミリ秒単位で日付を入力できるようにしても，あまり意味がありません．しかし，time_t型(システム日時を保存する型)のオブジェクトを受け取るようにオーバーロードするのであれば大変役に立ちます(このようなプログラムの例については，章末にある「この章の理解度チェック」とその解答を参照してください)．

**4.** クラスのコンストラクタ関数をオーバーロードしなければならない状況は，もう1つあります．それは，そのクラスの動的配列を割り当てるときです．前の章で説明したとおり，動的配列を初期化することはできません．したがって，初期値を受け取るコ

ンストラクタがある場合は，初期値を受け取らないコンストラクタもオーバーロードしなければなりません．次のプログラムでは，オブジェクト配列を動的に割り当てています．

```
#include <iostream>
using namespace std;

class myclass {
  int x;
public:
  // コンストラクタを2とおりにオーバーロードする
  myclass() { x = 0; }       // 初期化値を指定しない
  myclass(int n) { x = n; } // 初期化値を指定する
  int getx() { return x; }
  void setx(int n) { x = n; }
};

int main()
{
  myclass *p;
  myclass ob(10);       // 1つの変数を初期化する

  p = new myclass[10]; // ここでは初期値を使用できない
  if(!p) {
    cout << "Allocation error¥n";
    return 1;
  }

  int i;

  // すべての要素をobに初期化する
  for(i=0; i<10; i++) p[i] = ob;
  for(i=0; i<10; i++) {
    cout << "p[" << i << "]: " << p[i].getx();
    cout << '¥n';
  }

  return 0;
}
```

初期値を受け取らないmyclass()コンストラクタを用意しないと，new文によってコンパイルエラーが発生し，プログラムをコンパイルすることができません．

## 練習問題　5.1　コンストラクタ関数のオーバーロード

**1.** 次のような部分的なクラス定義があります．

```
class strtype {
  char *p;
  int len;
public:
  char *getstring() { return p; }
  int getlength() { return len; }
};
```

このクラス定義に，2つのコンストラクタ関数を追加しなさい．1つは仮引数を受け取らないようにします．このコンストラクタではnew演算子を使用して255バイトのメモリを割り当て，そのメモリをヌル文字列で初期化します．変数lenには255の値を格納します．もう一方のコンストラクタでは仮引数を2つ受け取ります．1つ目は初期化に使用する文字列，もう1つは割り当てるバイト数です．このコンストラクタでは指定された量のメモリを割り当て，文字列をそのメモリにコピーします．必要な境界チェックをすべて行い，短いプログラムを使用して，2つのコンストラクタの動作を確認しなさい．

**2.** 第2章の2.1節の例2で，ストップウォッチをまねたプログラムを作成しました．このプログラムを改良し，stopwatchクラスに仮引数なしのコンストラクタ(既存のコンストラクタ)とシステム時刻(標準関数clock()から返される時刻)を受け取るコンストラクタをオーバーロードしなさい．また，コンストラクタの動作を確認しなさい．

**3.** 自分のプログラミング作業の中で，コンストラクタ関数のオーバーロードが役に立つような状況を考えなさい．

# 5.2　コピーコンストラクタの作成と使用

オーバーロードコンストラクタの重要な用途として，**コピーコンストラクタ**(copy constructor)があります．これまでの章で紹介してきた多くのプログラムでは，オブジェクトを関数に渡したり，関数からオブジェクトを返す際に問題が発生することを述べてきました．これらの問題を避ける方法の1つが，この節で説明するコピーコンストラクタを定義する方法です．

まず，コピーコンストラクタを使用して解決する問題をここでもう一度確認しましょう．オブジェクトを関数に渡すと，そのオブジェクトのビット単位の(厳密な)コピーが作成され，オブジェクトを受け

取る関数仮引数として使用されます．しかし，まったく同じコピーを作成すべきではない状況もあります．たとえば，割り当てられたメモリへのポインタを含むオブジェクトがある場合，コピーオブジェクトは元のオブジェクトと同じメモリを指します．したがって，コピーオブジェクトでこのメモリの内容に変更を加えると，元のオブジェクトにも影響が及んでしまいます．また，関数の終了時には，コピーオブジェクトが破棄され，デストラクタ関数が呼び出されます．これによって，元のオブジェクトに望ましくない影響が及ぶ可能性があります．

関数からオブジェクトを返す場合にも，似たような状況が発生します．一般的なコンパイラは，関数から返す値のコピーを保持する一時オブジェクトを作成します(この処理は自動的に行われるので，プログラマが操作することはできません)．この一時オブジェクトは，呼び出し元のルーチンに値を返した後はスコープ外となり，一時オブジェクトのデストラクタが呼び出されます．もし呼び出し元ルーチンで必要な情報をデストラクタで破棄する(たとえば動的に割り当てたメモリを解放する)と，問題が発生します．

これらの問題点の原因となっているのは，オブジェクトのまったく同じコピーが作成されるということです．これらの問題を防ぐためには，望ましくない副作用が起こらないように，オブジェクトのコピーが作成されたときに実行する処理を正確に定義する必要があります．このために，コピーコンストラクタを作成します．コピーコンストラクタを定義することによって，オブジェクトのコピーを作成するときに実行する処理を正確に指定することができます．

C++では1つのオブジェクトを別のオブジェクトに与える状況が2種類あることを理解しておいてください．1つは代入，もう1つは初期化です．初期化は次の3とおりの場合に発生します．

- 宣言文で，オブジェクトを使用してほかのオブジェクトを宣言するとき
- 関数に仮引数としてオブジェクトを渡すとき
- 関数の戻り値として使用する一時オブジェクトを作成するとき

コピーコンストラクタでは，初期化の状況にしか対処できません．代入に対処することはできません．

デフォルトでは，初期化が発生すると，コンパイラは正確なコピーを自動的に作成します(つまり，C++によって，オブジェクトを単に複製するデフォルトコピーコンストラクタが自動的に用意されます)．しかし，コピーコンストラクタを定義すれば，1つのオブジェクトを使用してほかのオブジェクトを初期化する方法を正確に指定することができます．定義したコピーコンストラクタは，そのオブジェクトを使ってほかのオブジェクトを初期化する際に呼び出されます．

**重要**　コピーコンストラクタは，代入操作には効果がありません．

コピーコンストラクタの一般形式を次に示します．

**コピーコンストラクタ**

```
classname (const classname &obj) {
  // コンストラクタ本体
}
```

*obj*の部分は，ほかのオブジェクトを初期化するために使用するオブジェクトの参照です．例として，myclassというクラスがあり，yというmyclass型のオブジェクトがあるとしましょう．myclassコピーコンストラクタは，次の文を使用して呼び出すことができます．

```
myclass x = y; // yを使用してxを明示的に初期化する
func1(y);      // yを仮引数として渡す
y = func2();   // yを使用して返されたオブジェクトを受け取る
```

最初の2つの文では，コピーコンストラクタにyの参照を渡しています．3つ目の文では，func2()関数から返されたオブジェクトの参照を，コピーコンストラクタに渡しています．

## 例 5.2 コピーコンストラクタの作成と使用

**1.** 次のプログラムでは，明示的なコピーコンストラクタが必要となる理由を例示しています．このプログラムでは，きわめて制限された「安全な」整数配列型を作成し，配列境界を越えるのを防いでいます．new演算子を使用して各配列の記憶域を割り当て，各配列オブジェクト内でこのメモリへのポインタを管理しています．

```
/* このプログラムでは「安全な」配列クラスを作成する
   配列を記憶する領域を動的に割り当てるので，
   配列オブジェクトを使用してほかの配列オブジェクトを
   初期化する際に，コピーコンストラクタを使用して
   メモリを割り当てる
*/
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
  int *p;
  int size;
public:
  array(int sz) { // コンストラクタ
    p = new int[sz];
```

```
    if(!p) exit(1);
    size = sz;
    cout << "Using 'normal' constructor¥n";
  }
  ~array() {delete [] p;}

  // コピーコンストラクタ
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i] = j;
  }
  int get(int i) {
    return p[i];
  }
};

/* コピーコンストラクタ

   このコンストラクタでは，コピー用にメモリを割り当て，
   このメモリのアドレスをpに割り当てる
   したがって，pが元のオブジェクトと同じ動的割り当てメモリを
   参照することはない
*/
array::array(const array &a) {
  int i;

  size = a.size;
  p = new int[a.size];  // コピー用のメモリを割り当てる
  if(!p) exit(1);
  for(i=0; i<a.size; i++) p[i] = a.p[i]; // 内容をコピーする
  cout << "Using copy constructor¥n";
}

int main()
{
  array num(10);  // 「通常の」コンストラクタを呼び出す
  int i;

  // 配列に値を格納する
  for(i=0; i<10; i++) num.put(i, i);

  // numの内容を表示する
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "¥n";

  // ほかの配列を作成し，numを使用して初期化する
  array x = num;  // コピーコンストラクタを呼び出す
```

```
  // xを表示する
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

numを使ってxを初期化すると，コピーコンストラクタが呼び出され，新しい配列用のメモリが割り当てられてx.pに保存されます．numの内容はxの配列にコピーされます．この場合，xとnumには同じ値を持つ配列が保存されていますが，それぞれの配列はまったく別個のものです(つまり，num.pとx.pは同じメモリを指していません)．コピーコンストラクタを作成しなければ，x = numの初期化が正確に行われて，xとnumの配列が同じメモリを共有することになります(つまり，num.pとx.pがまったく同じ場所を指します)．

コピーコンストラクタは，初期化の際にだけ呼び出されます．たとえば，このプログラム内の次のプログラムコードでは，コピーコンストラクタは呼び出されません．

```
array a(10);
array b(10);

b = a; // コピーコンストラクタを呼び出さない
```

このb = aというプログラムコードは，代入を行っています．

**2.** コピーコンストラクタを使って，特定の型のオブジェクトを関数に渡す際に問題が発生するのを防ぐことを理解するために，次の例を考えてみましょう(これは正しくないプログラムの例です)．

```
// このプログラムにはエラーがある
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
```

```
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  strcpy(p, s);
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "¥n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

このプログラムでは，strtypeオブジェクトをshow()関数に渡す際に，厳密なコピーが作成され(コピーコンストラクタを定義していないため)，x仮引数として使われます．したがって，関数が返されるとxがスコープから外れ，破棄されます．これによって，当然ながらxのデストラクタが呼び出され，x.pが解放されます．しかし，解放されるメモリは，関数を呼び出すのに使用した元のオブジェクトが使用しているメモリと同じなので，エラーが発生します．

この問題の解決法は，strtypeクラスのコピーコンストラクタを定義し，コピーの作成時にコピー用のメモリを割り当てることです．次の修正版プログラムではこの方法を採用しています．

```
/* このプログラムでは，コピーコンストラクタを使用して
   strtypeオブジェクトを関数に渡す */
#include <iostream>
```

```
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);             // コンストラクタ
  strtype(const strtype &o);   // コピーコンストラクタ
  ~strtype() { delete [] p; } // デストラクタ
  char *get() { return p; }
};

// 「通常」のコンストラクタ
strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  strcpy(p, s);
}

// コピーコンストラクタ
strtype::strtype(const strtype &o)
{
  int l;

  l = strlen(o.p)+1;

  p = new char [l]; // 新しいコピーオブジェクトにメモリを割り当てる
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  strcpy(p, o.p); // 文字列をコピーオブジェクトにコピーする
}

void show(strtype x)
{
  char *s;
```

```
  s = x.get();
  cout << s << "¥n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

show()関数が終了してxがスコープから外れると，x.pが指すメモリが解放されますが，これは関数に渡した元のオブジェクトが使用しているメモリとは異なります．

## 練習問題 5.2 コピーコンストラクタの作成と使用

**1.** コピーコンストラクタは，関数の戻り値として使用する一時オブジェクトが作成される際にも呼び出されます(オブジェクトを返す関数の場合)．この点を踏まえて，次の出力について考えてみましょう．

```
Constructing normally
Constructing normally
Constructing copy
```

この出力は，次のプログラムによって生成されたものです．何が行われているかを正確に説明し，またその理由を説明しなさい．

```
#include <iostream>
using namespace std;

class myclass {
public:
  myclass();
  myclass(const myclass &o);
  myclass f();
};

// 通常のコンストラクタ
myclass::myclass()
{
  cout << "Constructing normally¥n";
```

```
}

// コピーコンストラクタ
myclass::myclass(const myclass &o)
{
  cout << "Constructing copy¥n";
}

// オブジェクトを返す
myclass myclass::f()
{
  myclass temp;
  return temp;
}

int main()
{
  myclass obj;
  obj = obj.f();
  return 0;
}
```

**2.**　次のプログラムの誤りを指摘し，それを修正しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int *p;
public:
  myclass(int i);
  ~myclass() { delete p; }
  friend int getval(myclass o);
};

myclass::myclass(int i)
{
  p = new int;

  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  *p = i;
```

```
}

int getval(myclass o)
{
  return *o.p; // 値を取得する
}

int main()
{
  myclass a(1), b(2);

  cout << getval(a) << " " << getval(b);
  cout << "¥n";
  cout << getval(a) << " " << getval(b);

  return 0;
}
```

**3.** コピーコンストラクタの目的と，通常のコンストラクタとの違いを，自分の言葉で説明しなさい．

## 5.3 古い overload キーワード

C++が開発された当時は，関数をオーバーロードするためにはoverloadキーワードを使わなければなりませんでした．今ではoverloadキーワードは必要なくなり，最近のC++コンパイラではサポートされていませんが，古いプログラムではこのキーワードが使われているのを見かけることがあります．そこで，overloadキーワードの使い方を学んでおくことにしましょう．

overloadの一般形式は次のとおりです．

overload

```
overload func-name;
```

func-nameの部分には，オーバーロードする関数の名前を指定します．この文は，オーバーロード関数を宣言する前に記述する必要があります．次の例では，timer()という名前の関数をオーバーロードすることをコンパイラに伝えています．

```
overload timer;
```

重要

overload は古いキーワードであり，最近の C++ コンパイラではサポートされていません．

## 5.4 デフォルト引数の使用

関数オーバーロードに関連する C++ 機能が1つあります．これは**デフォルト引数**(default argument)と呼ばれる機能で，これによって，関数の呼び出し時に指定されていない引数があるときに，対応する仮引数にデフォルト値を与えることができます．デフォルト引数は，関数オーバーロードの省略形のようなものです．

仮引数にデフォルト引数を指定するには，仮引数の後に等号を指定し，デフォルト値を指定するだけです．関数の呼び出し時に対応する引数が指定されていなければ，この値が使われます．次の関数では，2つの仮引数のデフォルトとして0を定義しています．

```
void f(int a=0, int b=0);
```

この構文は，変数の初期化の構文と似ています．これで，この関数は，3とおりの方法で呼び出すことができるようになりました．1つ目は両方の引数を指定して呼び出す方法です．2つ目は第1引数だけを指定して呼び出す方法です．この場合，b にはデフォルト値の0が使われます．3つ目は，引数をまったく指定せずに f() 関数を呼び出す方法です．この場合，a と b にはどちらもデフォルト値の0が使われます．つまり，次の f() 関数呼び出しはどれも有効です．

```
f();        // aとbにデフォルト値の0を使用する
f(10);      // aは10，bはデフォルト値の0
f(10, 99)  // aは10，bは99
```

この例からもわかるとおり，a にデフォルト値を使用し，b の値を指定することはできません．

1つまたは複数のデフォルト引数を持つ関数を作成する場合，これらの引数は1回しか指定できません．関数のプロトタイプか定義(関数を最初に使用する前に定義がある場合)のどちらかで指定します．プロトタイプと定義の両方でデフォルト値を指定することはできません．まったく同じデフォルト値を両方に指定することもできません．

おそらくお気付きでしょうが，すべてのデフォルト仮引数は，デフォルト値を持たないすべての仮引数よりも右側に記述する必要があります．さらに，デフォルト仮引数の定義を始めたら，それ以降にデフォルト値を持たない仮引数を指定することはできません．

また，デフォルト引数は定数かグローバル変数でなければなりません．ローカル変数やほかの仮引数をデフォルト引数として使うことはできません．

## 例 5.4　デフォルト引数の使用

**1.** 次のプログラムは，この節の説明で使用した例をコード化したものです．

```
// デフォルト引数の単純な例
#include <iostream>
using namespace std;

void f(int a=0, int b=0)
{
  cout << "a: " << a << ", b: " << b;
  cout << '¥n';
}

int main()
{
  f();
  f(10);
  f(10, 99);

  return 0;
}
```

このプログラムからの出力は，次のようになります．

```
a: 0, b: 0
a: 10, b: 0
a: 10, b: 99
```

1つ目のデフォルト引数を指定したら，それ以降の仮引数にもすべてデフォルト値を用意しなければならないことを覚えておいてください．たとえば，この関数を次のように修正すると，コンパイルエラーが発生します．

```
void f(int a=0, int b) // 誤り．bにもデフォルト値が必要
{
  cout << "a: " << a << ", b: " << b;
  cout << '¥n';
}
```

**2.** デフォルト引数と関数オーバーロードの関係を理解するために，2つのプログラムを比較してみましょう．1つ目のプログラムでは，rect_area()という関数をオーバーロードしています．この関数は，四角形の面積を返します．

```
// オーバーロード関数を使用して四角形の面積を計算する
#include <iostream>
using namespace std;
```

```
// 正方形ではない四角形の面積を返す
double rect_area(double length, double width)
{
  return length * width;
}

// 正方形の面積を返す
double rect_area(double length)
{
  return length * length;
}

int main()
{
  cout << "10 x 5.8 rectangle has area: ";
  cout << rect_area(10.0, 5.8) << '¥n';
  cout << "10 x 10 square has area: ";
  cout << rect_area(10.0) << '¥n';

  return 0;
}
```

このプログラムでは，rect_area()関数を2とおりにオーバーロードしています．1つ目の関数では，四角形の2辺のサイズを関数に渡します．この関数は，四角形が正方形ではない場合に使用します．しかし，四角形が正方形であるときは，引数を1つしか指定する必要がありません．この場合は2つ目のrect_area()関数が呼び出されます．

よく考えると，このような状況では，2つの異なる関数を用意する必要はないことがわかるはずです．2つ目の引数にデフォルト値を用意し，その値によってrect_area()関数の動作を変えればよいのです．第2引数にデフォルト値が使われた場合は，length仮引数を2回使用します．次にこの方法を採用したプログラムを示します．

```
// デフォルト引数を使用して四角形の面積を計算する
#include <iostream>
using namespace std;

// 四角形の面積を返す
double rect_area(double length, double width = 0)
{
  if(!width) width = length;
  return length * width;
}

int main()
{
  cout << "10 x 5.8 rectangle has area: ";
```

```
    cout << rect_area(10.0, 5.8) << '¥n';
    cout << "10 x 10 square has area: ";
    cout << rect_area(10.0) << '¥n';

    return 0;
  }
```

ここでは，width仮引数のデフォルト値として0を使用しています．この値を使用したのは，幅が0である四角形は存在しないからです(幅が0の四角形は，実際には線です)．したがって，width仮引数にデフォルト値が使われた場合，rect_area()関数ではwidth仮引数値としてlenght仮引数の値を自動的に使用します．

この例からわかるとおり，デフォルト引数は，関数オーバーロードの代わりとして簡単に利用できます(もちろん，関数オーバーロードが必要な場合も数多くあります)．

**3.** コンストラクタ関数にデフォルト引数を用意することもできます(実際には非常に一般的です)．この章で前述したとおり，初期化するオブジェクトと初期化しないオブジェクトの両方を作成できるようにするためだけに，コンストラクタをオーバーロードすることがよくあります．ほとんどの場合は，1つまたは複数のデフォルト引数を使用することによって，コンストラクタをオーバーロードしなくて済みます．例として，次のプログラムを見てみましょう．

```
#include <iostream>
using namespace std;
class myclass {
  int x;
public:
  /* myclassクラスのコンストラクタをオーバーロードする代わりに，
     デフォルト引数を使用する */
  myclass(int n = 0) { x = n; }
  int getx() { return x; }
};

int main()
{
  myclass o1(10); // 初期値を指定して宣言する
  myclass o2; // 初期値を指定せずに宣言する

  cout << "o1: " << o1.getx() << '¥n';
  cout << "o2: " << o2.getx() << '¥n';

  return 0;
}
```

この例では，nにデフォルト値として0を定義しているので，明示的な初期値を持つオブジェクトとデフォルト値を使用するオブジェクトを両方とも作成することができます．

**4.** このほかのデフォルト引数の用途として，仮引数を使用してオプションを選択する場合があります．仮引数にデフォルト値を用意し，そのデフォルト値を以前に選択されたオプションを引き続き使用することを示すフラグとして使うことができます．たとえば，次のプログラムのprint()関数は，画面に文字列を表示します．how仮引数にignoreが設定されている場合は，テキストをそのまま表示します．upperが設定されている場合は，テキストを大文字で表示します．lowerが設定されている場合は，テキストを小文字で表示します．how仮引数が設定されていない場合はデフォルト値の-1を使用し，前回使用したhow値を再び使用します．

```
#include <iostream>
#include <cctype>
using namespace std;

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

int main()
{
  print("Hello There¥n", ignore);
  print("Hello There¥n", upper);
  print("Hello There¥n"); // 大文字で表示を続ける
  print("Hello there¥n", lower);
  print("That's all¥n");  // 小文字で表示を続ける

  return 0;
}
/* 指定されたケースを使用して文字列を表示する
   指定されていない場合は前回のケースを使用する
*/
void print(char *s, int how)
{
  static int oldcase = ignore;

  // 指定されていない場合は前回のケースを使用する
  if(how<0) how = oldcase;
  while(*s) {
    switch(how) {
```

```
            case upper: cout << (char) toupper(*s);
              break;
            case lower: cout << (char) tolower(*s);
              break;
            default: cout << *s;
          }
          s++;
        }
        oldcase = how;
      }
```

この関数からの出力は次のようになります．

```
Hello There
HELLO THERE
HELLO THERE
hello there
that's all
```

**5.** この章で，前にコピーコンストラクタの一般形式を紹介しました．この一般形式では，仮引数が1つしかありませんでした．しかし，複数の引数を受け取るコピーコンストラクタを作成することもできます．ただし，これらの引数にはデフォルト値を用意する必要があります．たとえば，次の形式のコピーコンストラクタを使用することもできるのです．

```
myclass(const myclass &obj, int x=0) {
  // コンストラクタの本文
}
```

1つ目の引数がコピーするオブジェクトへの参照であり，ほかの引数がすべてデフォルト値を持っていれば，その関数はコピーコンストラクタとして扱われます．このような柔軟性のおかげで，ほかの用途を持つコピーコンストラクタを作成することもできます．

**6.** デフォルト引数は強力で便利な機能ですが，誤用されることもあります．正しい使い方をした場合は，関数の処理を効率的で扱いやすい方法で実行できることは間違いありません．しかし，これは仮引数にデフォルト値を与えるのが適切な場合だけです．たとえば，引数に10回中9回は使用する値がある場合は，デフォルト引数を使用するのがよいでしょう．しかし，特に多く使われる値がない場合や，フラグとしてデフォルト引数を使うことに利点がない場合は，デフォルト値を用意してもあまり意味がありません．実際には，不必要なデフォルト引数を使用すると，プログラムが破壊されたり，その関数を使用するほかのプログラマにとって誤解のもととなります．

優れたC++プログラマになるためには，関数オーバーロードと同様に，デフォルト引数を使用すべきときと使用すべきでないときを見極めることです．

## 練習問題　5.4　デフォルト引数の使用

**1.** C++標準ライブラリにはstrtol()という関数があります．この関数のプロトタイプは次のとおりです．

```
long strtol(const char *start, const **end, int base);
```

この関数は，startが指す数値文字列を長整数に変換します．baseには，数値文字列の基数を指定します．関数が返すと，endは文字列内の文字の数値のすぐ後を指し，数値文字列に相当する長整数が返されます．baseには2から38までの範囲で指定します．通常は10を使用します．

strtol()関数と同じ動作をするmystrtol()という名前の関数を作成しなさい．ただし，デフォルト引数として10を使用します(実際の変換処理を行う際には，strtol()関数を呼び出してもかまいません．そのためには<cstdlib>ヘッダを使用します)．また，作成した関数の動作を確認しなさい．

**2.** 次の関数プロトタイプの誤りを指摘しなさい．

```
char *f(char *p, int x = 0, char *q);
```

**3.** ほとんどのC++コンパイラには，カーソル位置を制御する非標準関数が用意されています．使用するコンパイラにそのような関数が用意されている場合は，myclreol()という関数を作成し，現在のカーソル位置から行末まで消去しなさい．この関数には，消去する文字位置を示す仮引数を用意します．この仮引数が指定されない場合は，行全体を自動的に消去します．指定された場合は，仮引数によって指定された数の文字位置だけを消去します．

**4.** デフォルト引数を使用した次のプロトタイプの誤りを指摘しなさい．

```
int f(int count, int max = count);
```

# 5.5　オーバーロードのあいまいさ

関数をオーバーロードすると，プログラムにあいまいさが生じることがあります．オーバーロードに起因するあいまいさは，型変換，参照仮引数，デフォルト引数などから発生します．さらに，オーバーロード関数そのものによって生じるあいまいさもあります．それ以外にも，オーバーロード関数の呼び出し方法によって発生するあいまいさがあります．あいまいさを取り除かないと，コンパイル時にエラーが発生します．

## 例　5.5　オーバーロードのあいまいさ

**1.** 最も多く発生するあいまいさは，C++の自動型変換規則に起因しています．渡した仮引数の型と互換性のある(同じではない)型を引数として関数に渡すと，引数の型は目的に型に自動的に変換されます．実際に，この種の変換のおかげで，putchar()など引数がint型で指定されている関数を，文字列を使って呼び出すことができます．しかし場合によっては，この自動型変換によって，関数をオーバーロードする際にあいまいさが生じることがあります．次にこのようなプログラムの例を示します．

```
// このプログラムにはあいまいさによるエラーがある
#include <iostream>
using namespace std;

float f(float i)
{
  return i / 2.0;
}

double f(double i)
{
  return i / 3.0;
}

int main()
{
  float x = 10.09;
  double y = 10.09;
  cout << f(x);  // あいまいではない．f(float)を使用する
  cout << f(y);  // あいまいではない．f(double)を使用する
  cout << f(10); // あいまい．10をdoubleとfloatのどちらにでも変換できる

  return 0;
}
```

main()関数のコメントに示したとおり，float型変数とdouble型変数を使用してf()関数を呼び出したときには，コンパイラはどの関数を選択すればよいかわかります．しかし，整数を渡したときはどうでしょう．f(float)とf(double)のどちらを呼び出せばよいのでしょうか(どちらも有効です)．どちらの場合も，整数をfloat型に変換することも，double型に変換することも可能です．したがって，あいまいな状況が発生します．

また，この例からわかるとおり，オーバーロード関数の呼び出し方法によってあいまいさが生じることもあります．あいまいではない引数を使用して呼び出す限り，オーバーロードのf()関数自体には，生来のあいまいさはありません．

**2.** 次のプログラムも，それ自体はあいまいではないオーバーロード関数の例です．この関数を呼び出すときに，誤った型の引数を渡すと，C++の自動変換規則のためにあいまいさが発生します．

```
// このプログラムにはあいまいさによるエラーがある
#include <iostream>
using namespace std;

void f(unsigned char c)
{
  cout << c;
}

void f(char c)
{
  cout << c;
}

int main()
{
  f('c');
  f(86); // どちらのf()関数を呼び出せばよいか？

  return 0;
}
```

数値定数86を使用してf()関数を呼び出すと，コンパイラはf(unsigned char)とf(char)のどちらを呼び出すべきか判断できません．どちらの変換も有効なので，あいまいさが生じます．

**3.** 参照仮引数を受け取るか，デフォルト値の値呼び出し仮引数を受け取るかという違いしかないオーバーロード関数を作成すると，あいまいさが発生します．C++の正式な

構文では，コンパイラはどちらの関数を呼び出すべきかを判断できません．値仮引数を受け取る関数と参照仮引数を受け取る関数は，どちらも呼び出し方法に構文上の違いがありません．次のプログラムを見てください．

```
// あいまいなプログラム
#include <iostream>
using namespace std;

int f(int a, int b)
{
  return a+b;
}

// 本質的にあいまいな関数
int f(int a, int &b)
{
  return a-b;
}

int main()
{
  int x=1, y=2;
  cout << f(x, y); // どちらのf()関数を呼び出せばよいか？
  return 0;
}
```

このプログラムでは，f(x, y)という呼び出しでどちらの関数でも呼び出すことができるため，あいまいさが生じます．実際には，2つの関数のオーバーロード自体があいまいであり，両者の参照を区別することができないので，f(x, y)という文を指定する以前にコンパイルエラーが発生します．

**4.** 1つまたは複数のオーバーロード関数でデフォルト引数を使用すると，あいまいさが生じることがあります．次のプログラムについて考えてみましょう．

```
// デフォルト引数とオーバーロードによるあいまいさ
#include <iostream>
using namespace std;

int f(int a)
{
  return a*a;
}

int f(int a, int b = 0)
{
  return a*b;
```

```
}

int main()
{
  cout << f(10, 2); // f(int, int)を呼び出す
  cout << f(10);    // あいまい．f(int)とf(int, int)のどちらを
                    // 呼び出せばよいか？

  return 0;
}
```

このプログラムで，f(10, 2)という文は完全に有効であり，あいまいさはありません．しかし，f(10)という文では，f()関数の1つ目と2つ目(変数bにデフォルト値を使用)のどちらを呼び出すべきかをコンパイラは判断できません．

**練習問題　5.5　オーバーロードのあいまいさ**

**1.** この節で紹介したあいまいなプログラムをコンパイルし，どのような誤りが生じるか確認しなさい．これは，実際に作成したプログラムにあいまいさがあったときに，その誤りを見分けるのに役立ちます．

## 5.6　オーバーロード関数のアドレスの探し方

この章の最後に，オーバーロード関数のアドレスの探し方を学びます．Cと同様に，関数のアドレス(エントリポイント)をポインタに割り当て，そのポインタを介して関数にアクセスすることができます．関数のアドレスを取得するには，代入文の右辺に，括弧や引数を付けずに関数名を記述します．たとえば，zap()という関数が正しく宣言されている場合，次のようにするとpにzap()関数のアドレスを代入することができます．

```
p = zap;
```

Cでは，指すことのできる関数が1つしかないので，あらゆる型のポインタを使用して関数を指すことができます．しかしC++では，関数をオーバーロードできることから，状況が少し異なります．つまり，アドレスを取得する関数を特定する機構が必要となります．

しかし，効果的ですばらしい解決法があります．オーバーロード関数のアドレスを取得する際には，ポインタの宣言方法によって，アドレスを取得するオーバーロード関数が決まります．つまり，ポインタの宣言がオーバーロード関数の宣言と比較されます．そして，一致する宣言を持つ関数のアドレスが使われます．

## 例 5.6 オーバーロード関数のアドレスの探し方

**1.** 次のプログラムでは，space()という関数を2とおりにオーバーロードしています．1つ目の関数では，count個の空白を画面に出力します．2つ目の関数では，chに渡された文字をcount個出力します．main()関数内では，2つの関数ポインタを宣言しています．1つ目は整数仮引数を1つだけ受け取る関数を指すポインタです．2つ目は2つの仮引数を受け取る関数を指すポインタです．

```
/* オーバーロード関数に関数ポインタを
   使用する例 */
#include <iostream>
using namespace std;

// count個の空白を出力する
void space(int count)
{
  for( ; count; count--) cout << ' ';
}

// count個のchを出力する
void space(int count, char ch)
{
  for( ; count; count--) cout << ch;
}

int main()
{
  /* 1つの整数仮引数を受け取るvoid関数への
     ポインタを作成する */
  void (*fp1)(int);

  /* 1つの整数仮引数と1つの文字仮引数を
     受け取るvoid関数へのポインタを作成する */
  void (*fp2)(int, char);

  fp1 = space;  // space(int)関数のアドレスを取得する
  fp2 = space;  // space(int, char)関数のアドレスを取得する

  fp1(22);        // 22個の空白を出力する
  cout << "|¥n";
  fp2(30, 'x'); // 30個のxを出力する
  cout << "|¥n";

  return 0;
}
```

コメントに示したとおり，コンパイラはfp1とfp2に宣言方法から，アドレスを取得するオーバーロード関数を判別することができます．

要点をまとめると，オーバーロード関数のアドレスを関数ポインタに代入する際には，どの関数のアドレスを代入するかは，ポインタの宣言によって決まります．さらに，関数ポインタの宣言は，いずれか1つのオーバーロード関数の宣言とまったく同じでなければなりません．そうでないと，あいまいさが生じ，コンパイルエラーが発生します．

### 練習問題　5.6　オーバーロード関数のアドレスの探し方

**1.** 次に，2つのオーバーロード関数を示します．それぞれのアドレスを取得する方法を示しなさい．

```
int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}
```

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** 5.1節の例3で紹介したdate()コンストラクタをオーバーロードし，time_t型の仮引数を受け取るようにしなさい(time_t型とは，C++コンパイラのライブラリにある標準時刻関数と標準日付関数によって定義される型です)．

**2.** 次のプログラムコードの誤りを指摘しなさい．

```
class samp {
  int a;
public:
  samp(int i) { a = i; }
```

```
    // ...
};

// ...

int main()
{
    samp x, y(10);

    // ...
}
```

**3.** クラスコンストラクタをオーバーロードする理由を2つ説明しなさい.

**4.** コピーコンストラクタの最も一般的な形式を示しなさい.

**5.** コピーコンストラクタを呼び出す原因となる操作について説明しなさい.

**6.** overloadキーワードの働きと，使われなくなった理由について簡単に説明しなさい.

**7.** デフォルト引数とは何か簡単に説明しなさい.

**8.** 2つの仮引数を受け取るreverse()という関数を作成しなさい．1つ目の仮引数(str)では文字列のポインタを受け取り，この関数の終了時にはこの文字列を逆順にして返します．2つ目の仮引数(count)では，逆順にする文字数を受け取ります．count仮引数に，文字列全体を逆順にするデフォルト値を用意しなさい.

**9.** 次のプロトタイプの誤りを指摘しなさい.

```
char *wordwrap(char *str, int size=0, char ch);
```

**10.** 関数をオーバーロードする際に，あいまいさが生じる状況をいくつか説明しなさい.

**11.** 次のプログラムコードの誤りを指摘しなさい.

```
void compute(double *num, int divisor=1);
void compute(double *num);
// ...
compute(&x);
```

**12.** オーバーロード関数のアドレスをポインタに代入する場合，アドレスを取得する関数がどのようにして判別されるか説明しなさい.

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** 2つの整数参照仮引数を受け取るorder()という関数を作成しなさい．1つ目の引数が2つ目の引数より大きい場合は，2つの引数を逆順にしなさい．それ以外の場合は何も行いません．つまり，order()関数に渡された2つの引数を並べ替え，終了時には1つ目の引数が2つ目の引数よりも小さくなるようにします．たとえば，次の場合は，関数が終了するとxには0，yには1が格納されるようにします．

```
int x=1, y=0;
order(x, y);
```

**2.** 次の2つのオーバーロード関数には本質的にあいまいさが潜んでいる理由を説明しなさい．

```
int f(int a);
int f(int &a);
```

**3.** デフォルト引数の使用が関数のオーバーロードに関係する理由を説明しなさい．

**4.** 次の不完全なクラスがあります．main()関数内の2つの宣言が有効となるようなコンストラクタ関数を作成しなさい(samp()関数を2とおりにオーバーロードする必要があります)．

```
class samp {
  int a;
public:
  // コンストラクタ関数を追加する
  int get_a() { return a; }
};

int main()
{
  samp ob(88);       // obのaを88に初期化する
  samp obarray[10]; // 初期化しない10要素の配列

  // ...
}
```

**5.** コピーコンストラクタが必要となる理由を簡単に説明しなさい．

# 6

# 演算子オーバーロード

## この章の内容

この章では，C++の重要な機能の1つである演算子のオーバーロードについて説明します．これは，クラスを定義するときに，そのクラスに関連付けてC++演算子の意味を定義する機能です．演算子をオーバーロードすることによって，プログラムに新しいデータ型を違和感なく追加できます．

## 前章の理解度チェック

この章を読み進む前に，次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

**1.** 初期化されていないオブジェクトを作成するには，次に示すクラスのコンストラクタをどのようにオーバーロードしたらよいか説明しなさい(初期化されていないオブジェクトを作成するには，x と y に値0を指定します)．

```
class myclass {
  int x, y;
public:
  myclass(int i, int j) { x=i; y=j; }
  //  ...
};
```

**2.** 問1のクラスで，デフォルト引数を使って，myclass()のオーバーロードを避けるにはどうすればよいか説明しなさい．

**3.** 次に示す宣言の誤りを指摘しなさい．

```
int f(int a=0, double balance);
```

**4.** 次に示すオーバーロードされた2つの関数の誤りを指摘しなさい．

```
void f(int a);
void f(int &a);
```

**5.** デフォルト引数を使用してかまわないのは，どのようなときか説明しなさい．また，使わない方がよいのは，どのようなときか説明しなさい．

**6.** 次のクラス定義があるとき，これらのオブジェクトの配列を動的に割り当てることは可能かどうか答えなさい．

```
class test {
  char *p;
  int *q;
```

```
  int count;
public:
  test(char *x, int *y, int c) {
    p = x;
    q = y;
    count = c;
  }
  //  ...
};
```

7. コピーコンストラクタとは，どういったもので，どのような状況で呼び出されるものか説明しなさい．

# 6.1　演算子オーバーロードの基本

演算子オーバーロードは，関数オーバーロードに似ています．実際には，関数オーバーロードの一種ですが，規則がいくつか加わります．たとえば，演算子をオーバーロードするときは，常にユーザーによって定義された何らかの型(クラスなど)に関連付けて定義しなければなりません．ほかにも違いがありますが，それについては随時説明していきます．

演算子をオーバーロードしても，その演算子が持つ本来の意味は失われません．むしろ，当該クラスにのみ適用される新しい意味が追加されることになります．

演算子をオーバーロードするには，演算子関数を作成します．ほとんどの場合，演算子関数は当該クラスのメンバかフレンドです．ただ，メンバ演算子関数とフレンド演算子関数には多少の違いがあるため，この章では先にメンバ演算子関数の作成を説明し，次にフレンド演算子関数を説明します．

メンバ演算子関数の一般形式を次に示します．

**メンバ演算子関数**

```
return-type class-name::operator#(arg-list)
{
    // 実行する演算
}
```

演算子関数の戻り型は，その演算子関数のオーバーロードの対象となっている当のクラスであることがよくあります(ただし，演算子関数はどのような型でも自由に返せます)．#の場所にはオーバーロードする演算子が入ります．たとえば，+をオーバーロードする場合は関数名はoperator+です．*arg-list*の内容は，演算子関数がどのように実装されているか，どの種類の演算子のオーバーロードかによって変わります．

演算子オーバーロードには，重要な制約が2つあります．1つは演算子の優先順序を変更できないこと，もう1つは演算子が受け取るオペランドの個数を変えられないことです．たとえば，/演算子をオーバーロードして，オペランド数を1つにすることはできません．

ほとんどのC++演算子はオーバーロードできます．オーバーロードできない演算子は次のとおりです．

**オーバーロードできない演算子**

```
. :: .* ?
```

また，プリプロセッサ演算子もオーバーロードできません(.*演算子は非常に特殊なので，本書では説明しません)．

C++では，さまざまな演算子が定義されていることを思い出してください．たとえば,[]添え字演算子,()関数呼び出し演算子，newとdelete，ドット(.)演算子とアロー(->)演算子などがあります．しかしこの章では，最もよく使われる演算子だけを取り上げ，そのオーバーロードを見ていくことにします．

すべての派生クラスは，=を除くすべての演算子関数を継承します．しかし，継承した演算子を(基本クラスですでにオーバーロードされている演算子も含め)，自分自身に関連付けて自由にオーバーロードすることができます．

これまでも，オーバーロードされた演算子を2つ使ってきました．それは<<と>>です．この演算子は，コンソール入出力を実行するようにオーバーロードされています．しかし，すでに説明したとおり，オーバーロードによって入出力を実行するようになっても，この演算子が本来持っている左シフトと右シフトの働きを失うことはありません．

演算子関数には，本来の動作に関連していようといまいと，どのような処理でも実行させることができます．しかし，本来の働きからあまり逸脱しない範囲でオーバーロードを行った方がよいでしょう．この原則からかけ離れたオーバーロードは，プログラム構造の実質的破壊の危険を冒すことになります．たとえば，/演算子オーバーロードによって「I like C++」という文をディスクファイルに300回書き込ませることは可能でしょうが，それは根本的に混乱をもたらすという意味で，演算子オーバーロードの誤用と言わざるをえません．

とはいうものの，本来の使い方とは関連のない方法で演算子を使用しなければならないことも，実際にあります．最も良い例が<<と>>です．これらの演算子にはオーバーロードによりコンソール入出力の働きを与えられています．しかし，この場合でも，左矢印と右矢印が，新しく定義された働きの視覚的「手掛かり」を与えるものになっています．オーバーロードによって演算子に標準的ではない意味を与えなければならない場合でも，できるだけ「ふさわしい」演算子を使うことが重要です．

最後に，演算子関数はデフォルト引数を持つことができません．

# 6.2　2項演算子のオーバーロード

メンバ演算子関数で2項演算子をオーバーロードするとき，その関数は仮引数を1つしか取りません．この仮引数は，演算子の右辺にあるオブジェクトを受け取ります．左辺のオブジェクトは，演算子関数への呼び出しを生成するオブジェクトで，thisによって暗黙的に引き渡されます．

演算子関数の書き方には，さまざまな形があることを理解しておくことが重要です．以下の例は決してあらゆる形を網羅しているとは言えませんが，最も良く使われる手法のいくつかを示しています．

## 例　6.2　2項演算子のオーバーロード

**1.** 次のプログラムでは，+演算子をcoordクラスに対してオーバーロードしています．このクラスは，X，Y座標の保持に使用されます．

```
// +をcoordクラスに対してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
};

// +をcoordクラスに対してオーバーロードする
coord coord::operator+(coord ob2)
{
  coord temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // 2つのオブジェクトを加算する．これはoperator+()を呼ぶ

  o3.get_xy(x, y);
```

```
    cout << "(o1+o2) X: " << x << ", Y: " << y << "¥n";

    return 0;
}
```

このプログラムからの出力を次に示します.

```
(o1+o2) X: 15, Y: 13
```

このプログラムをよく見てください. operator+()関数が返すcoord型のオブジェクトは, 各オペランドのX座標の合計をxに, Y座標の合計をyに持っています. operator+()では, 結果を入れておくためにtempという一時オブジェクトが使用され, 実際に返されるのはこのオブジェクトであることに注意してください. オペランド自体は, どちらも変更されません. tempを使用する理由は, すでに明らかでしょう. 上の例では(ほとんどの場合そうですが), +を通常の算術演算での用法と一貫するようにオーバーロードしています. したがって, どちらのオペランドも変更しないことが重要でした. たとえば, 10+4を計算すると結果は14ですが, 10も4も変更されません. したがって, 結果を入れておくための一時オブジェクトが必要となります.

operator+()関数がcoord型のオブジェクトを返すのは, coordオブジェクトどうしの加算結果をさらに大きな式の中で使用できるようにするためです. たとえば, 次の文を見てください.

```
o3 = o1 + o2;
```

この文は, o1+o2の結果がcoordオブジェクトであり, それをo3に代入できるからこそ有効なのです. 別の型が返されたのであれば, この文は無効になります. さらに, coordオブジェクトが返されるということは, 加算演算子によって連続加算ができることを意味します. たとえば, 次の文は有効です.

```
o3 = o1 + o2 + o1 + o3;
```

演算子関数に, 定義対象である当のオブジェクト以外の何かを返させたい状況があることは確かですが, 作成する演算子関数のほとんどが, 当該クラスのオブジェクトを返すのもまた事実です(例外として関係演算子と論理演算子のオーバーロードがありますが, それについては6.3節「関係演算子と論理演算子のオーバーロード」で学びます).

最後にもう一例見てみましょう. 返されるのがcoordオブジェクトなので, 次の文は完全に有効です.

```
(o1+o2).get_xy(x, y);
```

ここでは，operator+()によって返された一時オブジェクトが直接使用されています．もちろん，この文の実行後，その一時オブジェクトは破棄されます．

**2.** 次に示すのは前記プログラムの別バージョンです．ここでは，coordクラスの－演算子と＝演算子をオーバーロードしています．

```
// coordクラスの+，-，=をオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
  coord operator-(coord ob2);
  coord operator=(coord ob2);
};

// +をcoordクラスに対してオーバーロードする
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// -をcoordクラスに対してオーバーロードする
coord coord::operator-(coord ob2)
{
  coord temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// =をcoordクラスに対してオーバーロードする
coord coord::operator=(coord ob2)
```

```
{
  x = ob2.x;
  y = ob2.y;

  return *this; // 代入されたオブジェクトを返す
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // 2つのオブジェクトの加算．operator+()を呼ぶ
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "¥n";

  o3 = o1 - o2; // 2つのオブジェクトの減算
  o3.get_xy(x, y);
  cout << "(o1-o2) X: " << x << ", Y: " << y << "¥n";

  o3 = o1; // オブジェクトの代入
  o3.get_xy(x, y);
  cout << "(o3=o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

operator-()関数は，operator+()と同様に実装されます．しかし，上の例には，演算子オーバーロードにおけるオペランドの並び順という重要な点が示されています．operator+()関数を作成したときは，オペランドの並び順が問題になりませんでした(A+BはB+Aと同じです)．しかし，減算演算子は並び順で結果が変わります．減算演算子を正しくオーバーロードするには，左のオペランドから右のオペランドを引かなければなりません．operator-()への呼び出しを生成するのは左のオペランドなので，減算は次の並び順でなければなりません．

```
x - ob2.x;
```

**重要**　2項演算子のオーバーロードでは，左のオペランドは暗黙的に関数に渡され，右のオペランドは引数として渡されます．

次に，代入演算子関数を見てみましょう．最初に気付くことは，左辺のオペランド(値を代入されるオブジェクト)がこの演算によって変更されることです．これは，「代入」の一般的意味に一致しています．次に気付くのは，この関数が*thisを返すことです．

つまり，operator=()関数は，値を代入されたオブジェクトそのものを返します．その理由は，連続代入ができるようにするためです．C++では，次のような文も構文的に正しいと見なされ，実際によく使われています．

```
a = b = c = d = 0;
```

*this が返されるため，オーバーロードされた代入演算子ではcoord型のオブジェクトも同様の方法で使用できます．たとえば，次の文は完全に有効です．

```
o3 = o2 = o1;
```

実は，値の代入を受けたオブジェクトを返さなければならないという規則は，代入関数のオーバーロードにはありません．しかし，当該クラスに対してオーバーロードされた=の動作を，組み込み型に関する動作と同じにしたければ，オーバーロードされた=が*thisを返すようにしなければなりません．

**3.** 演算子関数をあるクラスに対してオーバーロードするとき，右辺のオペランドを，その演算子関数をメンバとするクラスでなく，組み込み型のオブジェクト(たとえば整数)にすることは可能です．たとえば，次の例に示す+演算子のオーバーロードでは，coordオブジェクトに整数値を加算しています．

```
// ob+obだけでなくob+intも行えるよう，+をオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2); // ob + ob
  coord operator+(int i); // ob+int
};

// +をcoordクラスに対してオーバーロードする
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}
```

```
// ob+intを行えるよう，+をオーバーロードする
coord coord::operator+(int i)
{
  coord temp;

  temp.x = x + i;
  temp.y = y + i;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // 2つのオブジェクトの加算．operator+(coord)を呼ぶ
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "¥n";

  o3 = o1 + 100; // オブジェクト+整数の加算．operator+(int)を呼ぶ
  o3.get_xy(x, y);
  cout << "(o1+100) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

メンバ演算子関数をオーバーロードして，組み込み型が関係する演算にオブジェクトも使用できるようにするときは，組み込み型を演算子の右側に置くようにしなければなりません．理由は簡単で，要するに，演算子関数への呼び出しを生成するのが左側のオブジェクトである，ということです．たとえば，コンパイラが次の文を見つけると何が起こるでしょうか．

```
o3 = 19 + o1;   //  int + ob
```

整数とオブジェクトの加算を扱うよう定義されている組み込み演算子はなく，オーバーロードされたoperator+(int i)関数は，オブジェクトが左側にあるときしか働きません．したがって，この文はコンパイルエラーになります(この制約の迂回方法を後述します)．

**4.** 演算子関数では参照仮引数を使用できます．たとえば，+演算子をcoordクラスに対して次のようにオーバーロードするのは，まったく問題がありません．

```
// 参照により，+をcoordクラスに対してオーバーロードする
coord coord::operator+(coord &ob2)
```

```
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}
```

演算子関数で参照仮引数を使用する理由の1つに，効率の良さがあります．オブジェクトを仮引数として関数に渡すと大量のオーバーヘッドを生じ，相当数のCPUサイクルを消費します．しかし，オブジェクトのアドレスの引き渡しは，常に高速かつ効率的です．使用頻度の高い演算子に参照仮引数を使用すると，一般にパフォーマンスが大幅に向上します．

参照仮引数を使用するもう1つの理由は，オペランドのコピーの破棄に伴う問題を避けられることです．これまでの各章で学んだとおり，引数を値で引き渡すと，その引数のコピーが作られます．そのオブジェクトにデストラクタ関数があると，関数終了時にコピーのデストラクタが呼ばれます．そのデストラクタが，呼び出しを行った側のオブジェクトにとって必要な何かまで破棄しないとも限りません．そのような場合でも，値仮引数でなく参照仮引数を使用すれば，問題を簡単に(かつ効率良く)迂回できます．もちろん，一般論としては，そのような問題を起こさないようにコピーコンストラクタを定義することが必要です．

## 練習問題　6.2　2項演算子のオーバーロード

**1.** * 演算子と / 演算子を coord に対してオーバーロードしなさい．正しく動作するか試してみなさい．

**2.** 次の例でオーバーロード演算子の使い方が不適当な理由を説明しなさい．

```
coord coord::operator%(coord ob)
{
  double i;

  cout << "Enter a number: ";
  cin >> i;
  cout << "root of " << i << " is ";
  cout << sqr(i);
}
```

**3.** 演算子関数の戻り型をcoord以外の何かに変更して，いろいろ試してみてください．どのようなエラーが生じるでしょうか．

## 6.3　関係演算子と論理演算子のオーバーロード

関係演算子と論理演算子もオーバーロードできます．関係演算子と論理演算子をオーバーロードしても，本来の動作をそのまま踏襲させたい場合は，定義対象の当のクラスのオブジェクトを演算子関数に返させるわけにはいきません．当然，真か偽を示す整数を返させることになります．こうすることで，演算子関数に真/偽の値を返させるだけでなく，その演算子を，ほかのデータ型も巻き込む大きな関係式や論理式の中に統合することができます．

メモ　最新のC++コンパイラでは，オーバーロードした関係演算子関数または論理演算子関数にbool型の値を返させることもできます(そうすることに，特に利点はありません)．第1章で説明したとおり，bool型では真(true)と偽(false)の2つの値しか定義されていません．この値は，自動的に非ゼロとゼロに変換されます．整数の非ゼロとゼロは，それぞれ自動的に真と偽に変換されます．

### 例　6.3　関係演算子と論理演算子のオーバーロード

**1.** 次のプログラムでは，== 演算子と && 演算子がオーバーロードされています．

```
// ==と&&をcoordクラスに対してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  int operator==(coord ob2);
  int operator&&(coord ob2);
};

// ==演算子をcoordに対してオーバーロードする
int coord::operator==(coord ob2)
{
```

```
  return x==ob2.x && y==ob2.y;
}

// &&演算子をcoordに対してオーバーロードする
int coord::operator&&(coord ob2)
{
  return (x && ob2.x) && (y && ob2.y);
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3(10, 10), o4(0, 0);

  if(o1==o2) cout << "o1 same as o2¥n";
  else cout << "o1 and o2 differ¥n";

  if(o1==o3) cout << "o1 same as o3¥n";
  else cout << "o1 and o3 differ¥n";

  if(o1&&o2) cout << "o1 && o2 is true¥n";
  else cout << "o1 && o2 is false¥n";

  if(o1&&o4) cout << "o1 && o4 is true¥n";
  else cout << "o1 && o4 is false¥n";

  return 0;
}
```

**練習問題 6.3 関係演算子と論理演算子のオーバーロード**

**1.** < 演算子と > 演算子を coord クラスに対してオーバーロードしなさい.

# 6.4 単項演算子のオーバーロード

単項演算子のオーバーロードは，2項演算子のオーバーロードに似ていますが，扱うオペランドが1つしかありません．メンバ関数を使用して単項演算子をオーバーロードする場合，その関数は仮引数を取りません．オペランドが1つしかないので，演算子関数への呼び出しを生成するのはそのオペランドです．ほかに仮引数は必要ありません．

## 例 6.4　単項演算子のオーバーロード

**1.** 次のプログラムでは，インクリメント演算子(++) をcoordクラスに対してオーバーロードしています．

```
// ++をcoordクラスに対してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator++();
};

// ++をcoordクラスに対してオーバーロードする
coord coord::operator++()
{
  x++;
  y++;

  return *this;
}

int main()
{
  coord o1(10, 10);
  int x, y;

  ++o1; // オブジェクトのインクリメント
  o1.get_xy(x, y);
  cout << "(++o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

インクリメント演算子はオペランドに1を加えるように設計されているため，オーバーロードされた++も，演算対象のオブジェクトを変更します．また，インクリメントされたオブジェクトを返します．このため，インクリメント演算子は，より大きな文の一部としても使用できます．次に例を示します．

```
o2 = ++o1;
```

2項演算子の場合と同様，単項演算子のオーバーロードでも，その本来の意味を尊重しなければならないという規則はありません．しかし，ほとんどの場合は，尊重するのが無難でしょう．

**2.** インクリメント演算子またはデクリメント演算子をオーバーロードするとき，初期のC++では，オーバーロードされた++または--がオペランドの前に置かれているのか，後ろに置かれているのかを知る方法がありませんでした．したがって，上のプログラムでは，次のどちらの文を使っても違いはありませんでした．

```
o1++;
++o1;
```

しかし，最新のC++仕様では，コンパイラがこの2つの文を識別する方法が定義されています．そのためには，operator++()関数を2とおり作成します．1つは，上の例に示したとおりに定義します．もう1つは，次のように宣言します．

```
coord coord::operator++(int notused);
```

++がオペランドの前に置かれていると，operator++()関数が呼ばれます．しかし，++がオペランドの後ろに置かれていると，operator++(int notused)関数が呼ばれます．この場合，notusedには常に値0が渡されます．したがって，インクリメントまたはデクリメントの前置と後置の違いが重要な意味を持つクラスオブジェクトでは，両方の演算子関数を実装する必要があります．

**3.** C++では，負符号が2項演算子としても単項演算子としても使用されます．クラスを作成する際に，この2とおりの使い方をそのまま保ちながら負符号をオーバーロードするにはどうすればいいか，迷っている方もいるのではないでしょうか．解決策は実は非常に簡単です．オーバーロードを二度行えばよいのです．一度は2項演算子，もう一度は単項演算子としてオーバーロードします．次に例を示します．

```
// -をcoordクラスに対してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator-(coord ob2); // 2項負符号
```

```
  coord operator-(); // 単項負符号
};

// -をcoordクラスに対してオーバーロードする
coord coord::operator-(coord ob2)
{
  coord temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// 単項-をcoordクラスに対してオーバーロードする
coord coord::operator-()
{
  x = -x;
  y = -y;

  return *this;
}

int main()
{
  coord o1(10, 10), o2(5, 7);
  int x, y;

  o1 = o1 - o2; // 減算
  o1.get_xy(x, y);
  cout << "(o1-o2) X: " << x << ", Y: " << y << "¥n";

  o1 = -o1; // 負にする
  o1.get_xy(x, y);
  cout << "(-o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

ご覧のとおり，2項演算子としてオーバーロードされた負符号は，仮引数を1つ取ります．単項演算子としてオーバーロードされた負符号は，仮引数を取りません．このように仮引数の数が違うからこそ，負符号をどちらの演算用にもオーバーロードできるわけです．上のプログラムでは，負符号を2項演算子として使用すると，operator-(coord ob2)関数が呼び出されます．単項演算子として使用すると，operator-()関数が呼び出されます．

## 練習問題　6.4　単項演算子のオーバーロード

1. -- 演算子を coord クラスに対してオーバーロードしなさい．前置形式と後置形式の両方を作成してみてください．

2. + 演算子を，2 項演算子としても（これまでどおり）と単項演算子としても使えるように，coord クラスに対してオーバーロードしなさい．単項演算子としての+には，負の座標値をすべて正の座標値に変える働きを持たせます．

# 6.5　フレンド演算子関数の使用

この章の冒頭で説明したとおり，演算子をあるクラスに対してオーバーロードするとき，メンバ関数ではなくフレンド関数を使用することもできます．フレンド関数はthisポインタを持ちません．したがって，2 項演算子では，2 つのオペランドがともに明示的にフレンド演算子関数に渡されます．単項演算子では，1 つのオペランドが渡されます．ほかの条件が同じであれば，一般に，メンバ演算子関数でなくフレンド演算子関数をわざわざ使用する理由はありませんが，1 つだけ重要な例外があります．それについては，次の例の中で触れることにします．

**重要**　代入演算子のオーバーロードにはフレンドを使用できません．代入演算子は，メンバ演算子関数でしかオーバーロードできません．

## 例　6.5　フレンド演算子関数の使用

1. 次のプログラムでは，operator+() をcoordクラスに対してオーバーロードするのに，フレンド関数を使用しています．

```
// フレンド関数を使用し，+をcoordクラスに対してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
```

```
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator+(coord ob1, coord ob2);
};

// フレンド関数を使用して+をオーバーロードする
coord operator+(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x + ob2.x;
  temp.y = ob1.y + ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // 2つのオブジェクトの加算. operator+()を呼ぶ
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

左側のオペランドが最初の仮引数，右側のオペランドが2番目の仮引数に渡されることに注意してください.

**2.** 演算子のオーバーロードにフレンドを使用すると，メンバ関数を使った場合には得られない重要な機能が1つ得られます．つまり組み込み型が関係する演算において，その組み込み型を演算子の左側に使用できるようになります．この章ではこれまで，演算子の左側のオペランドがオブジェクトで，右側のオペランドが組み込み型であるようなオーバーロードは，2項メンバ演算子関数を用いてもできることを見てきました．しかし，組み込み型が演算子の左側に来るようなオーバーロードは，メンバ関数ではできません．たとえば，メンバ演算子関数を使ったとすると，次に示す最初の文は正しく，2番目の文は誤りです.

```
ob1 = ob2 + 10; // 正しい
ob1 = 10 + ob2; // 誤り
```

すべての文を最初の形にすることはできますが，オブジェクトが必ず演算子の左側に置かれ，組み込み型が必ず右側に置かれるよう，いつも目を光らせているのはなかな

か疲れます．この問題を解決するには，オーバーロードする演算子関数をフレンドにして，両方の場合を定義してください．

フレンド演算子関数には，2つのオペランドがともに明示的に引き渡されます．したがって，まず1つのフレンド関数をオーバーロードして，左側のオペランドがオブジェクトで，右側のオペランドがほかの型であるようにします．次に，もう一度演算子をオーバーロードして，今度は左側のオペランドが組み込み型，右側のオペランドがオブジェクトになるようにします．次のプログラムでは，そのようにしています．

```
// フレンド演算子関数を使用して，多少融通をきかせる
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator+(coord ob1, int i);
  friend coord operator+(int i, coord ob1);
};

// ob+intのオーバーロード
coord operator+(coord ob1, int i)
{
  coord temp;

  temp.x = ob1.x + i;
  temp.y = ob1.y + i;

  return temp;
}
// int+obのオーバーロード
coord operator+(int i, coord ob1)
{
  coord temp;

  temp.x = ob1.x + i;
  temp.y = ob1.y + i;

  return temp;
}

int main()
{
```

```
  coord o1(10, 10);
  int x, y;

  o1 = o1 + 10; // オブジェクト+整数
  o1.get_xy(x, y);
  cout << "(o1+10) X: " << x << ", Y: " << y << "¥n";

  o1 = 99 + o1; // 整数+オブジェクト
  o1.get_xy(x, y);
  cout << "(99+o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

どちらにも使用できるようフレンド演算子関数をオーバーロードした結果，次の文はともに有効になりました．

```
o1 = o1 + 10;
o1 = 99 + o1;
```

**3.** ++単項演算子または--単項演算子のオーバーロードにフレンド演算子関数を使用したいときは，オペランドを参照仮引数として関数に渡さなければなりません．これは，フレンド関数がthisポインタを持たないためです．インクリメント演算子とデクリメント演算子では，必然的にオペランドが変更されることを思い出してください．しかし，値仮引数を使用するフレンドによってこの演算子をオーバーロードすると，フレンド演算子関数の中で仮引数に生じる変更が，呼び出しを生成したオブジェクトに影響を及ぼすことを避けられます．フレンド使用時には，暗黙的にオブジェクトに渡されるポインタがないので(つまり，thisポインタがないので)，インクリメントまたはデクリメントによりオペランドが変化することもなくなります．

しかし，オペランドを参照仮引数としてフレンドに渡すときは，フレンド関数の中で生じる変更が，呼び出しを生成するオブジェクトに影響を及ぼします．たとえば，次に示すプログラムでは，フレンド関数を用いて++演算子をオーバーロードしています．

```
// フレンド関数を使って++をオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
```

```
  friend coord operator++(coord &ob);
};

// フレンド関数を使って++をオーバーロードする
coord operator++(coord &ob) // 参照仮引数を使用
{
  ob.x++;
  ob.y++;

  return ob; // 呼び出しを生成するオブジェクトを返す
}

int main()
{
  coord o1(10, 10);
  int x, y;

  ++o1;  // o1は，参照によって引き渡される
  o1.get_xy(x, y);
  cout << "(++o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

最新のコンパイラを使用しているときは，メンバ関数使用時とほぼ同じ方法でフレンド演算子関数を使い，インクリメント演算子またはデクリメント演算子の前置形式と後置形式も識別することができます．後置形式を定義するときに，整数仮引数を1つ追加してください．たとえば，次に示すのは，coordクラスに関するインクリメント演算子の前置形式と後置形式のプロトタイプです．

```
coord operator++(coord &ob); // 前置
coord operator++(coord &ob, int notused); // 後置
```

++がオペランドの前にあると，operator++(coord &ob)関数が呼ばれます．しかし，++がオペランドの後ろにあると，operator++(coord &ob,int notused)関数が呼ばれます．この場合，notusedには値0が渡されます．

## 練習問題　6.5　フレンド演算子関数の使用

**1.** フレンド関数を使用し，－演算子と / 演算子をcoordクラスに対してオーバーロードしなさい．

2. 整数値に各座標値を掛ける演算を行う演算にcoordオブジェクトを使用できるよう，coordクラスをオーバーロードしなさい．ob*intとint*obのどちらも扱えるようにしなさい．

3. 練習問題2の解答で，なぜフレンド演算子関数を使用する必要があるのかを説明しなさい．

4. フレンドを使用し，-- をcoordクラスに対してどうオーバーロードするか示しなさい．前置形式も後置形式も定義しなさい．

## 6.6　代入演算子の詳細

すでに見たとおり，代入演算子も，特定クラスに対してオーバーロードすることができます．デフォルトでは，代入演算子をオブジェクトに適用すると，右辺のオブジェクトのビット単位コピーが左辺のオブジェクトに代入されます．この演算で十分なら，独自のoperator=()関数を定義する必要などありません．しかし，厳密なビット単位コピーが望ましくない場合もあります．第3章でいくつかの例を見たとおり，オブジェクトがメモリを割り当てる場合には，特殊な代入演算子を用意するとよいでしょう．

### 例　6.6　代入演算子の詳細

1. これまでの各章で，さまざまな形式のstrtypeクラスを見てきました．ここでもう1つ別のバージョンを見ておきましょう．このバージョンでは，代入演算によってポインタpが上書きされないように=演算子をオーバーロードしています．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *s);
  ~strtype() {
     cout << "Freeing " << (unsigned) p << '¥n';
     delete [] p;
```

```
    }
    char *get() { return p; }
    strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  len = l;
  strcpy(p, s);
}

// オブジェクトを代入する
strtype &strtype::operator=(strtype &ob)
{
  // さらにメモリが必要か調べる
  if(len < ob.len) { // さらにメモリの割り当てが必要
    delete [] p;
    p = new char [ob.len];
    if(!p) {
      cout << "Allocation error¥n";
      exit(1);
    }
  }
  len = ob.len;
  strcpy(p, ob.p);
  return *this;
}

int main()
{
  strtype a("Hello"), b("There");

  cout << a.get() << '¥n';
  cout << b.get() << '¥n';

  a = b; // これで，pは上書きされない

  cout << a.get() << '¥n';
```

```
    cout << b.get() << '¥n';

    return 0;
  }
```

ご覧のとおり，オーバーロードされた代入演算子は，pの上書きを防ぎます．まず，左辺のオブジェクトに，代入される文字列を保持できるだけのメモリが割り当られているかどうかがチェックされます．十分でないときは，そのメモリが解放され別のメモリ部分が割り当てられます．その後，文字列がそのメモリにコピーされ，長さがlenにコピーされます．

operator=()関数については，あと2つ重要な特徴があります．1つは，参照仮引数を取ることです．これにより，代入演算子の右辺のオブジェクトのコピーは作られません．すでに各章で学んだとおり，関数への引き渡し時にオブジェクトのコピーが作成されると，そのコピーは関数の終了とともに破棄されます．このとき，コピーの破棄のためにデストラクタ関数が呼び出され，pが解放されます．しかしこのpは，引数に使用されるオブジェクトがまだ必要としているpです．参照仮引数を使用すれば，この問題が避けられます．

operator=()関数のもう1つの重要な特徴は，オブジェクトではなく参照を返すことです．この理由は，参照仮引数を使用する理由と同じです．関数がオブジェクトを返すときには一時オブジェクトが作成され，返し終わるとともに破棄されます．しかし，そのために一時オブジェクトのデストラクタが呼ばれ，pが解放されます．しかし，このp（とそれが指し示すメモリ）は，値の代入を受けたオブジェクトがまだ必要としているpです．参照を返すことにすれば，一時オブジェクトを作成せずに済みます．

**！重要**

第5章で学んだとおり，ここで説明した問題は，2つともコピーコンストラクタを作成することで防ぐことができます．しかし，コピーコンストラクタによる解決は，参照仮引数と参照戻り型を使用する方法に比べて効率の点で劣ります．参照を使用すれば，上記どちらの場合でも，オブジェクトのコピーに伴うオーバーヘッドを回避できます．このように，C++では，同じ目的を達成するのにいくとおりもの方法があることがあります．良い方法を選ぶ能力も，優れたC++プログラマの資格の1つです．

**練習問題 6.6 代入演算子の詳細**

1. 次のクラス宣言が与えられたとき，動的配列型を作成するために必要な細部を埋めなさい．つまり，配列にメモリを割り当て，このメモリへのポインタをpに格納し，配列のサイズをバイト数でsizeに格納しなさい．put()には，指定された要素への参照を返させます．get()には，指定された要素の値を返させます．配列の範囲を超えることがあってはなりません．また，代入演算子をオーバーロードし，1つの配列が別の配列に代入されたとき，各配列に割り当てられているメモリが不注意で破棄されることがないようにしてください(次の節で，この練習問題の解答を改良する方法を説明します)．

```
class dynarray {
  int *p;
  int size;
public:
  dynarray(int s); // 配列のサイズをsに渡す
  int &put(int i); // 要素iへの参照を返す
  int get(int i);  // 要素iの値を返す
  // operator=()関数を作成する
};
```

# 6.7 []添え字演算子のオーバーロード

最後にオーバーロードする演算子は，[]配列添え字演算子です．C++では，[]はオーバーロードの目的に使う2項演算子と見なされています．[]は，メンバ関数でしかオーバーロードできません．メンバoperator[]()関数の一般形式は次のとおりです．

**operator[]()関数**

```
type class-name::operator[ ](int index)
{
   //  ...
}
```

技術的には，仮引数はint型でなくてもかまいません．しかし，operator[]()関数は，通常，配列の添え字に使用されるものなので，一般には整数値が使用されます．

[]演算子の働き方を理解するため，Oというオブジェクトが次のように添え字指定されていると想定しましょう．

```
O[9]
```

この添え字は，次のoperator[]()関数呼び出しに変換されます．

```
O.operator[](9)
```

つまり，添え字演算子内の式の値は，明示的仮引数を通じてoperator[]()関数に渡されます．thisポインタは，呼び出しを生成したオブジェクトであるOを指し示します．

## 例 6.7 []添え字演算子のオーバーロード

**1.** 次のプログラムでは，arraytypeが5個の整数からなる配列を宣言し，コンストラクタ関数が配列の各メンバを初期化しています．オーバーロードされたoperator[]()関数は，仮引数によって指定された要素の値を返します．

```
#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype {
  int a[SIZE];
public:
  arraytype() {
    int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  int operator[](int i) { return a[i]; }
};

int main()
{
  arraytype ob;
  int i;

  for(i=0; i<SIZE; i++)
    cout << ob[i] << " ";

  return 0;
}
```

このプログラムの出力は次のようになります．

```
0 1 2 3 4
```

このプログラムと次に示すプログラムでは，コンストラクタで配列aを初期化していますが，これはあくまでも例として示しているだけで，初期化しなくてもかまいません．

**2.** operator[]()関数の設計次第では，代入文の左辺と右辺の両方に[]を使用することも可能です．それには，添え字で指し示された要素への参照を返します．たとえば，次のプログラムで必要な変更を行っています．使い方を見てください．

```
#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype {
  int a[SIZE];
public:
  arraytype() {
    int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  int &operator[](int i) { return a[i]; }
};

int main()
{
  arraytype ob;
  int i;

  for(i=0; i<SIZE; i++)
    cout << ob[i] << " ";
  cout << "¥n";

  // 配列の各要素に10を加算する
  for(i=0; i<SIZE; i++)
    ob[i] = ob[i]+10;  // =の左辺に[]

  for(i=0; i<SIZE; i++)
    cout << ob[i] << " ";

  return 0;
}
```

このプログラムの出力は次のようになります．

```
0 1 2 3 4
10 11 12 13 14
```

このoperator[]()関数は，iによって添え字指定された配列要素への参照を返します．

したがって，これを代入演算子の左辺に使用して，配列の要素を変更することができます(もちろん，右辺でも使用できます)．ご覧のとおり，この方法によって，arraytypeのオブジェクトを通常の配列のように動作させることができます．

**3.** []演算子をオーバーロードできることの利点の1つは，配列に安全な添え字指定ができることです．本書では，以前にも安全な配列を実装する簡単な方法を見ましたが，それはget()関数やput()関数を使って配列の要素にアクセスする方法でした．次に説明する方法は，オーバーロードされた[]演算子を利用して安全な配列を作ろうというもので，こちらの方が優れています．安全な配列とは，境界チェックを行うクラスの中にカプセル化されている配列であることを思い出してください．この方法では，配列が境界を越えることがありません．[]演算子のオーバーロードによって作成した配列には，通常の配列と変わりなくアクセスできます．

安全な配列を作成するには，operator[]()関数に境界チェックを付け加えます．また，このoperator[]()関数は，添え字指定された要素への参照も返さなければなりません．たとえば，次のプログラムは，すでに示した配列プログラムに範囲チェック機能を付け加えたものです．境界エラーが発生することから，正しく機能することがわかります．

```
// 安全な配列の例
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 5;

class arraytype {
  int a[SIZE];
public:
  arraytype() {
    int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  int &operator[](int i);
};

// arraytypeの範囲チェック
int &arraytype::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
    cout << "¥nIndex value of ";
    cout << i << " is out of bounds.¥n";
    exit(1);
```

```
    }
    return a[i];
  }

  int main()
  {
    arraytype ob;
    int i;

    // これはOK
    for(i=0; i<SIZE; i++)
      cout << ob[i] << " ";
    /* SIZE+100は範囲外
       実行時エラーになる */
    ob[SIZE+100] = 99; // エラー!

    return 0;
  }
```

このプログラムで次の文が実行されると，

```
ob[SIZE+100] = 99;
```

境界エラーがoperator[]()によって捕獲され，何も問題が起こらないうちにプログラムが打ち切られます．

[]演算子のオーバーロードを使えば，外見も動作も通常の配列と変わりない安全配列を作成し，使用しているプログラミング環境に自然に統合できます．しかし，一点だけ注意が必要です．安全な配列にはオーバーヘッドが伴うため，どんな場合でも受け入れられるとは限りません．そもそも，C++が配列の境界チェックを行わないのは，このオーバーヘッドが問題であるためです．しかし，どうしても境界エラーを発生させたくないアプリケーションでは，安全な配列を試してみるとよいでしょう．その価値はあります．

## 練習問題 6.7 []添え字演算子のオーバーロード

**1.** 6.6節の例1を修正し，strtypeで[]演算子をオーバーロードします．この演算子が指定の添え字を持つ文字を返せるようにしなさい．また,[]を代入文の左辺で使用できるようにしてください．できあがったら，実際に試してみてください．

**2.** 6.6節の練習問題1の解答を修正して,[]によって動的配列の添え字指定ができるようにしなさい．つまり，get()関数とput()関数を[]演算子で置き換えます．

## この章の理解度チェック

この段階で，次の質問に回答できるかどうか確認しましょう．

**1.** >>と<<のシフト演算子をcoordクラスに対してオーバーロードし，次の種類の演算を行えるようにしなさい．

```
ob << integer
ob >> integer
```

その演算子がx値とy値を指定量だけシフトさせることを確認しなさい．

**2.** 次のクラスがあります．

```
class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
     x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
};
```

+，-，++，--各演算子をこのクラスに対してオーバーロードしなさい(インクリメント演算子とデクリメント演算子については，前置形式のみオーバーロードしなさい)．

**3.** 問2の答えを書き直し，演算子関数の仮引数を値仮引数でなく参照仮引数にしなさい
（ヒント）インクリメント演算子とデクリメント演算子には，フレンド関数を使用します．

**4.** フレンド演算子関数は，メンバ演算子関数とどのように違うか説明しなさい．

**5.** 代入演算子をオーバーロードしなければならないのはどのような場合か説明しなさい．

**6.** operator=()はフレンド関数でありうるかどうか説明しなさい．

**7.** +を問2のthree_dクラスに対してオーバーロードし，次の種類の演算も受け入れられるようにしなさい．

```
ob + int;
int + ob;
```

**8.** ==, !=, || の各演算子を問2のthree_dクラスに対してオーバーロードしなさい．

**9.** []演算子をオーバーロードする主な理由を説明しなさい．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** 次の演算ができるようなstrtypeクラスを作成しなさい．

- ＋演算子による文字列連結
- ＝演算子による文字列代入
- <, >, == による文字列比較

固定長文字列を使ってもかまいません．難問ですが，よく考えていろいろと試してみてください．必ずできるはずです．

# 7

# 継承

## この章の内容

継承という概念は，本書でも以前に紹介しました．この章では，それをもう少し詳しく見ていきます．継承はオブジェクト指向プログラミング(OOP)の3原則の1つであり，C++の重要な特徴でもあります．継承は，単に階層的クラス分けの概念をサポートするだけではありません．やはりOOPの主要な特徴であるポリモーフィズムをサポートしますが，それについては第10章で学ぶことにします．

この章では，基本クラスのアクセス制御とprotectedアクセス指定子，複数の基本クラスの継承，基本クラスコンストラクタへの引数の引き渡し，仮想基本クラスなどのトピックを取り上げます．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

**1.** オーバーロードされた演算子は，本来の機能を失うかどうか答えなさい．

**2.** 演算子のオーバーロードでは，それを何らかのユーザー定義型(たとえば，クラス)と関連付けて定義することが必要かどうか説明しなさい．

**3.** オーバーロードされた演算子の優先順位は変更できるかどうか説明しなさい．オペランドの数は変更できるかどうか説明しなさい．

**4.** 次に示すプログラムは未完成です．必要な演算子関数を補いなさい．

```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator+(array ob2);
  array operator-(array ob2);
  int operator==(array ob2);
};

array::array()
{
  int i;
```

```
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "¥n";
}

// 演算子関数を補う

int main()
{
  array o1, o2, o3;
  int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 equals o2¥n";
  else cout << "o1 does not equal o2¥n";

  if(o1==o3) cout << "o1 equals o3¥n";
  else cout << "o1 does not equal o3¥n";

  return 0;
}
```

+のオーバーロードでは，各オペランドの各要素を加算するようにしなさい．-のオーバーロードでは，左側のオペランドの各要素から右側のオペランドの各要素を引くよ

うにしなさい．==のオーバーロードでは，両オペランドの各要素が同じとき真を返し，それ以外のときは，偽を返すようにしなさい．

**5.** 問4の答えを手直しし，今度はフレンド関数を用いて演算子をオーバーロードしなさい．

**6.** 問4のクラスとサポート関数を使用し，++演算子をメンバ関数で，--演算子をフレンドでオーバーロードしなさい(++と--の前置形式だけをオーバーロードしなさい)．

**7.** 代入演算子をフレンド関数でオーバーロードすることはできるかどうか説明しなさい．

## 7.1　基本クラスのアクセス制御

あるクラスが別のクラスを継承するとき，次の一般形式が使われます．

**クラスの継承**

```
class derived-class-name : access base-class-name {
  //  ...
}
```

*access*には，3つのキーワードpublic，private，protectedのいずれかを指定します．protectedアクセス指定子については，次の節で説明することにして，ここではほかの2つを説明します．

アクセス指定子は，基本クラスの諸要素が派生クラスにどう継承されるかを決定します．継承される基本クラスのアクセス指定子が**public**であると，基本クラスのすべての公開メンバが派生クラスの公開メンバになります．アクセス指定子が**private**であると，基本クラスのすべての公開メンバが派生クラスの非公開メンバになります．どちらの場合でも，基本クラスの非公開メンバは基本クラスだけのものであり，派生クラスからはアクセスできません．

アクセス指定子が**private**のとき，基本クラスの公開メンバは派生クラスの非公開メンバになります．しかし，このメンバは，派生クラスのメンバ関数からはアクセスできるので，間違えないでください．これを理解しておくことは重要です．

技術的には，*access*の指定は任意です．指定子が明示されないと，デフォルト指定子が適用されます．派生クラスがclassのときはprivateがデフォルト，派生クラスがstructのときはpublicがデフォルトです．ほとんどのプログラマは，プログラムのわかりやすさを重視して*access*を明示的に指定します．

## 例 7.1 基本クラスのアクセス制御

**1.** 次に簡単な基本クラスと，それをpublicとして継承する派生クラスを示します．

```
#include <iostream>
using namespace std;

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '¥n'; }
};

// publicとして継承する
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '¥n'; }
};

int main()
{
  derived ob;

  ob.setx(10); // 基本クラスのメンバにアクセス
  ob.sety(20); // 派生クラスのメンバにアクセス
  ob.showx();  // 基本クラスのメンバにアクセス
  ob.showy();  // 派生クラスのメンバにアクセス

  return 0;
}
```

このプログラムでは，baseがpublicとして継承されているので，baseの公開メンバであるsetx()とshowx()はderivedの公開メンバになり，プログラムのどの部分からもアクセスできます．つまり，main()内から正当に呼び出すことができます．

**2.** 基本クラスをpublicとして継承しても，それは，派生クラスが基本クラスの非公開メンバにアクセスできることを意味するものではありません．この点をよく理解しておいてください．たとえば，問1の例のderivedに次のような追加を行うことは誤りです．

```
class base {
  int x;
public:
  void setx(int n) { x = n; }
```

```
  void showx() { cout << x << '¥n'; }
};

// publicとして継承．エラーあり！
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }

  /* 基本クラスの非公開メンバにはアクセスできない．
     xは基本クラスの非公開メンバであり，
     派生クラス内では使用できない．*/
  void show_sum() { cout << x+y << '¥n'; } // エラー！

  void showy() { cout << y << '¥n'; }
};
```

この例では，derivedクラスがbaseの非公開メンバxにアクセスしようとしています．これはエラーです．基本クラスの非公開部分は，継承方法にかかわらず非公開のままです．

**3.** 次に示すのは，例1のプログラムを少し変更したものです．今度は，derivedがbaseをprivateとして継承しています．この変更によりプログラムにエラーが発生します．コメントを見てください．

```
// このプログラムにはエラーがある
#include <iostream>
using namespace std;

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '¥n'; }
};

// 基本クラスをprivateとして継承する
class derived : private base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '¥n'; }
};

int main()
{
  derived ob;
```

```
    ob.setx(10); // エラー - 派生クラスに対して非公開
    ob.sety(20); // 派生クラスのメンバにアクセス - OK

    ob.showx();  // エラー - 派生クラスに対して非公開
    ob.showy();  // 派生クラスのメンバにアクセス - OK

    return 0;
  }
```

この(誤った)プログラムのコメントにあるとおり，showx()とsetx()はともにderivedに対して非公開となり，外部からのアクセスはできません.

showx()とsetx()は，派生クラスにどう継承されようと，base内部では依然として公開であることを忘れないでください．つまり，base型のオブジェクトは，どこにあってもこれらの関数にアクセスできます．しかし，derived型のオブジェクトに対しては，どちらの関数も非公開になります．たとえば，次のプログラムコードを見てください.

```
base base_ob;
base_ob.setx(1); // base_obはbase型なので正しい
```

setx()はbase内では公開なので，このsetx()呼び出しは正しいです.

**4.** すでに述べたとおり，継承時にprivate指定子が使われ，その結果，基本クラスの公開メンバが派生クラスの非公開メンバになったとしても，派生クラスの内部ではそのメンバにアクセスできます．たとえば，次に示すのは例3のプログラムの修正版です.

```
// このプログラムは修正済み
#include <iostream>
using namespace std;

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '¥n'; }
};

// 基本クラスをprivateとして継承する
class derived : private base {
  int y;
public:
  // 派生クラス内からはsetxにアクセスできる
  void setxy(int n, int m) { setx(n); y = m; }
  // 派生クラス内からはshowxにアクセスできる
```

```
  void showxy() { showx(); cout << y << '¥n'; }
};

int main()
{
  derived ob;

  ob.setxy(10, 20);
  ob.showxy();

  return 0;
}
```

この場合，関数setx()とshowx()は，派生クラスの内部でアクセスされています．このクラスの非公開メンバなので，完全に正しいです．

## 練習問題 7.1 基本クラスのアクセス制御

**1.** 次のスケルトンを見てください．

```
#include <iostream>
using namespace std;

class mybase {
  int a, b;
public:
  int c;
  void setab(int i, int j) { a = i; b = j; }
  void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
//  ...
};

class derived2 : private mybase {
//  ...
};

int main()
{
  derived1 o1;
  derived2 o2;
  int i, j;
```

```
            //  ...
        }
```

main()内で使用するには，次のどの文が正しいか判断しなさい．

```
A.  o1.getab(i, j);
B.  o2.getab(i, j);
C.  o1.c = 10;
D.  o2.c = 10;
```

2. 公開メンバをpublicとして継承するとどうなるか，また，公開メンバをprivateとして継承したらどうなるか説明しなさい．

3. この節の例を，まだ実行していなければ，実行しなさい．アクセス指定子をさまざまに変えて，その結果を調べなさい．

## 7.2　被保護メンバの使用

前節で学んだとおり，派生クラスからは基本クラスの非公開メンバにアクセスできません．派生クラスから基本クラスのメンバにアクセスする必要があるときは，そのメンバを公開メンバにしておかなければなりません．しかし，ときには，基本クラスのメンバを非公開にしたままで，派生クラスからそれにアクセスできれば都合がよいことがあります．それをできるようにするのが，C++に用意されている**protected**(被保護)アクセス指定子です．

protectedアクセス指定子はprivate指定子とほぼ同じですが，1つだけ異なる点があります．それは，基本クラスの被保護メンバには，その基本クラスから派生したどのクラスのメンバからもアクセスできる，ということです．基本クラスと派生クラス以外からは，被保護メンバにアクセスできません．

protectedアクセス指定子は，クラス宣言のどこに指定してもかまいませんが，通常は(デフォルトによる)非公開メンバ宣言の後ろ，公開メンバ宣言の前に置かれます．クラス宣言の完全な一般形式を次に示します．

**クラス宣言の一般形式**

```
class class-name {
    // 非公開メンバ
protected:   // 指定は任意
    // 被保護メンバ
public:
    // 公開メンバ
```

基本クラスの被保護メンバがpublicとして派生クラスに継承されると，それは派生クラスの被保護メンバになります．基本クラスがprivateとして継承されると，その基本クラスの被保護メンバは派生クラスの非公開メンバになります．

基本クラスが，protectedとして派生クラスに継承されることもあります．この場合，その基本クラスの公開メンバと被保護メンバは，派生クラスの被保護メンバになります(もちろん，基本クラスの非公開メンバは派生クラスに対して非公開のままなので，派生クラスからはアクセスできません)．

protectedアクセス指定子は，構造体でも使用できます．

## 例 7.2　被保護メンバの使用

**1.** 次のプログラムは，クラスの公開・非公開・被保護の各メンバへのアクセス方法を示しています．

```
#include <iostream>
using namespace std;

class samp {
  // デフォルトにより非公開
  int a;
protected: // samp以外には非公開
  int b;
public:
  int c;
  samp(int n, int m) { a = n; b = m; }
  int geta() { return a; }
  int getb() { return b; }
};

int main()
{
  samp ob(10, 20);

  // ob.b = 99; エラー! bは被保護．したがって非公開
  ob.c = 30; // OK．cは公開

  cout << ob.geta() << ' ';
  cout << ob.getb() << ' ' << ob.c << '¥n';

  return 0;
}
```

ご覧のとおり，コメント化されている行は，main()の中では使用できません．bは被保護なので，samp以外では非公開です．

**2.** 次のプログラムは，被保護メンバがpublicとして継承されたとき何が起こるかを示しています．

```
#include <iostream>
using namespace std;

class base {
protected:  // 基本クラス以外には非公開ながら
  int a, b; // 派生クラスからはアクセスできる
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
  int c;
public:
  void setc(int n) { c = n; }
  // この関数は，基本クラスのaとbにアクセスできる
  void showabc() {
    cout << a << ' ' << b << ' ' << c << '¥n';
  }
};

int main()
{
  derived ob;

  /* aとbは，ここではアクセスできない.
     基本クラスとその派生クラス以外には非公開 */
  ob.setab(1, 2);

  ob.setc(3);

  ob.showabc();

  return 0;
}
```

aとbは，base内でprotectedであり，publicとしてderivedに継承されています．したがって，derivedのメンバ関数からは使用できますが，その2つのクラス以外では実質的に非公開であり，アクセスできません．

**3.** 前述のとおり，基本クラスをprotectedとして継承すると，その基本クラスの公開メンバと被保護メンバは，派生クラスで被保護メンバになります．たとえば，次に示すプログラムでは上のプログラムを多少変更し，baseをpublicでなくprotectedとして継承しています．

```
// このプログラムはコンパイルされない
#include <iostream>
using namespace std;

class base {
protected:     // 基本クラス以外では非公開ながら
  int a, b;    // 派生クラスからはアクセスできる
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : protected base { // protectedとして継承
  int c;
public:
  void setc(int n) { c = n; }
  // この関数は基本クラスのaとbにアクセスできる
  void showabc() {
    cout << a << ' ' << b << ' ' << c << '¥n';
  }
};

int main()
{
  derived ob;

  // エラー：setab()は基本クラスの被保護メンバ
  ob.setab(1, 2); // ここからはsetab()にアクセスできない

  ob.setc(3);

  ob.showabc();

  return 0;
}
```

コメントにあるとおり，baseがprotectedとして継承されているので，その公開要素と被保護要素はderivedの被保護メンバとなり，main()内ではアクセスできません．

## 練習問題　7.2　被保護メンバの使用

**1.** 被保護メンバをpublicとして継承するとどうなるか説明しなさい．また，privateとして継承するとどうなるか，さらに，protectedとして継承するとどうなるか説明しなさい．

**2.**　なぜ被保護というカテゴリが必要なのか，説明しなさい．

**3.**　7.1 節の練習問題 1 で，mybase 内の a と b をデフォルトによる非公開メンバとせず，被保護メンバにすると，答えが変わりますか．変わるとしたら，どう変わるのかを説明しなさい．

# 7.3　コンストラクタ，デストラクタ，継承

基本クラスにも派生クラスにも，または必要ならその両方に，コンストラクタ関数やデストラクタ関数を含めることができます．ここでは，これらの関数にかかわる問題をいくつか取り上げます．

基本クラスと派生クラスの両方がコンストラクタ関数とデストラクタ関数を含んでいる場合，コンストラクタ関数は派生順に実行され，デストラクタ関数はその逆の順序で実行されます．つまり，まず基本クラスのコンストラクタが実行され，次に派生クラスのコンストラクタが実行されます．デストラクタ関数の実行順序はその逆です．先に派生クラスのデストラクタが実行され，次に基本クラスのデストラクタが実行されます．

コンストラクタ関数が派生順に実行されるのは，合理的であることがおわかりでしょう．基本クラスには，派生クラスについての知識がないため，そこで行われる初期化は，派生クラスで実行される初期化とは別物で，場合によってはそれの前提にもなるでしょう．したがって，これを最初に実行しなければなりません．

一方，デストラクタの実行順序が逆になるのは，派生クラスが基本クラスという土台の上にのっているからです．先に基本クラスのデストラクタを実行したのでは，派生クラスが破壊されてしまいます．したがって，まず派生クラスのデストラクタを呼び，オブジェクトそのものがなくなってしまう前に実行しておかなければなりません．

これまでに見てきた例の中には，派生クラスまたは基本クラスのコンストラクタに引数を渡す例がありませんでしたが，それも可能です．派生クラスだけを初期化をするのであれば，通常の方法で派生クラスのコンストラクタに引数を渡せますが，基本クラスのコンストラクタに引数を渡す必要があるときは，少し作業が増えます．これを行うには，引数渡しの連鎖を確立しておかなければなりません．まず，基本クラスと派生クラスの両方に必要な引数を，派生クラスのコンストラクタに全部渡します．次に，派生クラスのコンストラクタ宣言の拡張形式を使用して，該当する引数を基本クラスに引き渡します．派生クラスから基本クラスへ引数を渡す構文は，次のとおりです．

派生クラスから基本クラスへの引数の引き渡し

```
derived-constructor (arg-list) : base(arg-list) {
    // 派生クラスコンストラクタの本体
}
```

上に示す構文の*base*は基本クラスの名前です．派生クラスと基本クラスがともに同じ引数を使用しても差し支えありません．派生クラスがすべての引数を無視し，それを単に基本クラスに引き渡すだけのこともあります．

## 例 7.3　コンストラクタ，デストラクタ，継承

**1.** 次に示す短いプログラムを見ると，基本クラスと派生クラスのコンストラクタ関数，またデストラクタ関数が，いつ実行されるかがわかります．

```
#include <iostream>
using namespace std;

class base {
public:
  base() { cout << "Constructing base class¥n"; }
  ~base() { cout << "Destructing base class¥n"; }
};

class derived : public base {
public:
  derived() { cout << "Constructing derived class¥n"; }
  ~derived() { cout << "Destructing derived class¥n"; }
};

int main()
{
  derived o;
  return 0;
}
```

このプログラムの出力は次のようになります．

```
Constructing base class
Constructing derived class
Destructing derived class
Destructing base class
```

ご覧のとおり，コンストラクタは派生順序どおりに実行され，デストラクタは逆の順序で実行されます．

**2.** 次のプログラムは，派生クラスのコンストラクタに引数を渡す方法を示しています．

```
#include <iostream>
using namespace std;

class base {
public:
  base() { cout << "Constructing base class¥n"; }
  ~base() { cout << "Destructing base class¥n"; }
};

class derived : public base {
  int j;
public:
  derived(int n) {
    cout << "Constructing derived class¥n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class¥n"; }
  void showj() { cout << j << '¥n'; }
};

int main()
{
  derived o(10);

  o.showj();

  return 0;
}
```

引数が，通常の方法で派生クラスのコンストラクタに渡されていることに注意してください．

**3.** 次のプログラムでは，派生クラスのコンストラクタと基本クラスのコンストラクタがともに引数を取ります．ここでは両方が同じ引数を使用し，派生クラスは単にその引数を基本クラスに渡すだけの働きしかしません．

```
#include <iostream>
using namespace std;

class base {
  int i;
```

```
public:
  base(int n) {
    cout << "Constructing base class¥n";
    i = n;
  }
  ~base() { cout << "Destructing base class¥n"; }
  void showi() { cout << i << '¥n'; }
};

class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // 引数を基本クラスに渡す
    cout << "Constructing derived class¥n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class¥n"; }
  void showj() { cout << j << '¥n'; }
};

int main()
{
  derived o(10);

  o.showi();
  o.showj();

  return 0;
}
```

特に注目すべきは，派生クラスのコンストラクタ宣言です．仮引数n（これが初期化引数を受け取ります）はderived()で使用されるとともに，base()にも渡されています．

**4.** 基本クラスのコンストラクタ関数と派生クラスのコンストラクタ関数が同じ引数を使用するということはまれです．両者が異なる引数を取り，それぞれに1つ以上の引数を渡す必要があるときは，まず，派生クラスが取る引数と基本クラスが取る引数の両方を派生クラスのコンストラクタに渡さなければなりません．そして，そのうち基本クラスが必要とする引数を，派生クラスから基本クラスに渡すようにします．たとえば，次のプログラムでは，派生クラスのコンストラクタに1つの引数を渡し，基本クラスにも別の引数を渡しています．

```
#include <iostream>
using namespace std;

class base {
```

```
  int i;
public:
  base(int n) {
    cout << "Constructing base class¥n";
    i = n;
  }
  ~base() { cout << "Destructing base class¥n"; }
  void showi() { cout << i << '¥n'; }
};

class derived : public base {
  int j;
public:
  derived(int n, int m) : base(m) { // 引数を基本クラスに渡す
    cout << "Constructing derived class¥n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class¥n"; }
  void showj() { cout << j << '¥n'; }
};

int main()
{
  derived o(10, 20);

  o.showi();
  o.showj();

  return 0;
}
```

**5.** 派生クラスから基本クラスに引数を渡す上で，派生クラスのコンストラクタが実際に何らかの引数を使用しなければならないということはありません．派生クラスに引数が必要なければ，派生クラスは引数を無視し，それを基本クラスに渡すだけでかまいません．たとえば，次のプログラムコードでは，derived()は仮引数nを使用しません．それをbase()に渡すだけです．

```
class base {
  int i;
public:
  base(int n) {
    cout << "Constructing base class¥n";
    i = n;
  }
  ~base() { cout << "Destructing base class¥n"; }
  void showi() { cout << i << '¥n'; }
};
```

```
class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // 引数を基本クラスを渡す
    cout << "Constructing derived class¥n";
    j = 0; // ここではnを使用しない
  }
  ~derived() { cout << "Destructing derived class¥n"; }
  void showj() { cout << j << '¥n'; }
};
```

## 練習問題　7.3　コンストラクタ，デストラクタ，継承

**1.** 次のスケルトンにmyderivedのコンストラクタ関数を書き込みなさい．このコンストラクタ関数はmybaseに初期化文字列へのポインタを渡します．また，myderived()が文字列の長さに合わせてlenを初期化するようにしなさい．

```
#include <iostream>
#include <cstring>
using namespace std;

class mybase {
  char str[80];
public:
  mybase(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

class myderived : public mybase {
  int len;
public:
  // ここにmyderived()を追加する
  int getlen() { return len; }
  void show() { cout << get() << '¥n'; }
};

int main()
{
  myderived ob("hello");

  ob.show();
  cout << ob.getlen() << '¥n';

  return 0;
}
```

**2.** 次のスケルトンを使って，car()コンストラクタ関数とtruck()コンストラクタ関数を作成しなさい．どちらも適切な引数をvehicleに渡します．また，オブジェクト作成時にはcar()が指定どおりpassengersを初期化し，truck()が指定どおりloadlimitを初期化するようにしなさい．

```
#include <iostream>
using namespace std;

// 各種車両の基本クラス
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Wheels: " << num_wheels << '¥n';
    cout << "Range: " << range << '¥n';
  }
};

class car : public vehicle {
  int passengers;
public:
  // car()コンストラクタをここに挿入
  void show()
  {
    showv();
    cout << "Passengers: " << passengers << '¥n';
  }
};

class truck : public vehicle {
  int loadlimit;
public:
  // truck()コンストラクタをここに挿入
  void show()
  {
    showv();
    cout << "loadlimit " << loadlimit << '¥n';
  }
};
```

```
int main()
{
  car c(5, 4, 500);
  truck t(30000, 12, 1200);

  cout << "Car: ¥n";
  c.show();
  cout << "¥nTruck:¥n";
  t.show();

  return 0;
}
```

car()とtruck()では，次のようにオブジェクトを宣言します．

```
car ob(passengers, wheels, range);
truck ob(loadlimit, wheels, range);
```

## 7.4　多重継承

派生クラスは複数の基本クラスを継承できます．その方法は2とおりあります．まず，ある派生クラスを基本クラスとして，そこからさらに別の派生クラスを作成する方法です．この方法では，複数のレベルからなるクラス階層が出来上がります．この場合，最初の基本クラスを2番目の派生クラスの**間接基本クラス**(indirect base class）と呼びます(どのように作成したどのクラスでも，基本クラスとして使用できることを覚えておいてください)．もう1つは，派生クラスが複数の基本クラスを直接継承する方法です．この方法では，2つ以上の基本クラスが結合されて，そこから派生クラスが生成されます．複数の基本クラスが関係すると，それに伴っていくつかの問題が生じます．ここでは，その問題を見ていくことにします．

ある基本クラスから派生クラスを導出し，その派生クラスを基本クラスとしてさらに別の派生クラスを導出する場合，その3つのクラスのコンストラクタ関数は，派生の順序に従って呼び出されます(この章ですでに学んだ原則を一般的に表現すると，こうなります)．また，デストラクタ関数は逆の順序で呼ばれます．したがって，クラスB1がD1に継承され，D1がD2に継承されるとき，最初にB1のコンストラクタが呼ばれ，次にD1のコンストラクタ，その次にD2のコンストラクタが呼ばれます．デストラクタは，この逆の順序で呼び出されます．

派生クラスが複数の基本クラスを直接継承するときは，次の拡張宣言が使用されます．

継承の拡張宣言

```
class derived-class-name : access base1, access base2, . . . ,
access baseN
{
    // クラスの本体
}
```

*base1*から*baseN*は基本クラス名，*access*はアクセス指定子です．アクセス指定子は基本クラスごとに異なっていてかまいません．複数の基本クラスが継承されるとき，コンストラクタは，基本クラスの指定順序に従って左から右へ実行されます．デストラクタは，反対の順序で実行されます．

あるクラスが複数の基本クラスを継承し，その基本クラスのコンストラクタが引数を要求している場合，派生クラスは，次に示す拡張形式の派生クラスコンストラクタ関数を用いて，各基本クラスに必要な引数を渡します．

派生クラスコンストラクタ関数

```
derived-constructor(arg-list) : base1(arg-list), base2(arg-list),
. . . ,
    baseN(arg-list)
{
    // 派生クラスコンストラクタの本体
}
```

base1からbaseNは，基本クラスの名前です．

派生クラスがクラス階層を継承するときは，その階層中の各派生クラスが，それぞれの(直前にある)基本クラスに，同基本クラスが必要とする引数を渡さなければなりません．

## 例 7.4　多重継承

1. 次に示すのは，あるクラスから派生したクラスを，さらに別の派生クラスが継承している例です．D2からB1まで，継承の連鎖の中を引数がどう渡されていくか注意してください．

```
// 多重継承
#include <iostream>
using namespace std;

class B1 {
  int a;
```

```
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// 直接基本クラスを継承する
class D1 : public B1 {
  int b;
public:
  D1(int x, int y) : B1(y) // B1にyを渡す
  {
    b = x;
  }
  int getb() { return b; }
};

// 派生クラスと間接基本クラスを継承する
class D2 : public D1 {
  int c;
public:
  D2(int x, int y, int z) : D1(y, z) // D1に引数を渡す
  {
    c = x;
  }

  /* 2つの基本クラスがpublicとして継承されているため,
     D2はB1とD1両方の公開要素にアクセスできる */
  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '¥n';
  }
};
int main()
{
  D2 ob(1, 2, 3);

  ob.show();
  // geta()とgetb()は, 依然, 公開のまま
  cout << ob.geta() << ' ' << ob.getb() << '¥n';

  return 0;
}
```

ob.show()を呼び出すと，3 2 1と表示されます．この例では，B1がD2の間接基本クラスです．D2は，D1とB1の両方の公開メンバにアクセスできることに注意してください．基本クラスの公開メンバがpublicとして継承されると，そのまま派生クラ

スの公開メンバになることは，すでに学んだとおりです．したがって，D1がB1を継承すると，geta()はD1の公開メンバになり，さらにD2の公開メンバになります．

クラス階層を構成する各クラスが，それぞれ直前の基本クラスに必要なすべての引数を渡さなければならないことも，プログラムから見てとれます．これをしないと，コンパイルエラーになります．

このプログラムで作成されているクラス階層は，次のようになります．

B1

D1

D2

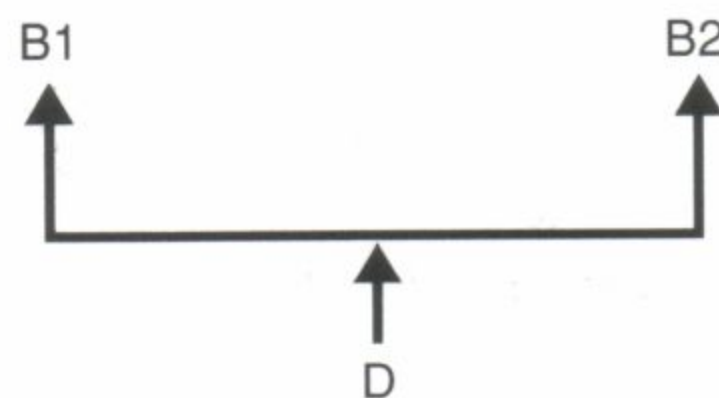
先に進む前に，C++流の継承グラフの描き方を簡単に説明しておきましょう．この図では，矢印が下ではなく上を向いていることに注意してください．C++プログラマは継承図を描くとき，習慣として，矢印の向きが派生クラスから基本クラスを指すような有向グラフを描きます．初めて見る人は直感的でないと思うかもしれませんが，C++では通常このようにして継承図を描きます．

**2.** 次に示すのは，例1のプログラムを書き直したものです．派生クラスが2つの基本クラスを直接継承しています．

```
#include <iostream>
using namespace std;

// 1つ目の基本クラスを作成
class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// 2つ目の基本クラスを作成
class B2 {
  int b;
```

```
public:
  B2(int x)
  {
    b = x;
  }
  int getb() { return b; }
};

// 2つの基本クラスを直接継承
class D : public B1, public B2 {
  int c;
public:
  // zとyをB1とB2に直接渡す
  D(int x, int y, int z) : B1(z), B2(y)
  {
    c = x;
  }

  /* 2つの基本クラスがpublicとして継承されたため,
     DはB1とB2の両方の公開要素にアクセスできる */
  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '¥n';
  }
};

int main()
{
  D ob(1, 2, 3);

  ob.show();

  return 0;
}
```

このバージョンでは，B1とB2への引数が，Dからこれらのクラスへ個々に渡されます．このプログラムで作成されるクラス階層は，次のようになります．

B1　　B2
↑　　↑
D

**3.**　次のプログラムでは，派生クラスが複数の基本クラスを直接継承したとき，コンストラクタ関数とデストラクタ関数がどのような順序で呼び出されるかがわかります．

```
#include <iostream>
using namespace std;

class B1 {
public:
  B1() { cout << "Constructing B1¥n"; }
  ~B1() { cout << "Destructing B1¥n"; }
};

class B2 {
  int b;
public:
  B2() { cout << "Constructing B2¥n"; }
  ~B2() { cout << "Destructing B2¥n"; }
};

// 2つの基本クラスを継承
class D : public B1, public B2 {
public:
  D() { cout << "Constructing D¥n"; }
  ~D() { cout << "Destructing D¥n"; }
};

int main()
{
  D ob;

  return 0;
}
```

このプログラムの出力は次のようになります．

```
Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

すでに学んだとおり，複数の直接基本クラスが継承されるとき，コンストラクタは継承リストに指定されている順序どおり，左から右へ呼び出されます．デストラクタは，逆の順序で呼ばれます．

## 練習問題 7.4 多重継承

**1.** 次のプログラムは何を表示するか説明しなさい(プログラムを実行せずに答えなさい).

```
#include <iostream>
using namespace std;

class A {
public:
  A() { cout << "Constructing A¥n"; }
  ~A() { cout << "Destructing A¥n"; }
};

class B {
public:
  B() { cout << "Constructing B¥n"; }
  ~B() { cout << "Destructing B¥n"; }
};

class C : public A, public B {
public:
  C() { cout << "Constructing C¥n"; }
  ~C() { cout << "Destructing C¥n"; }
};

int main()
{
  C ob;

  return 0;
}
```

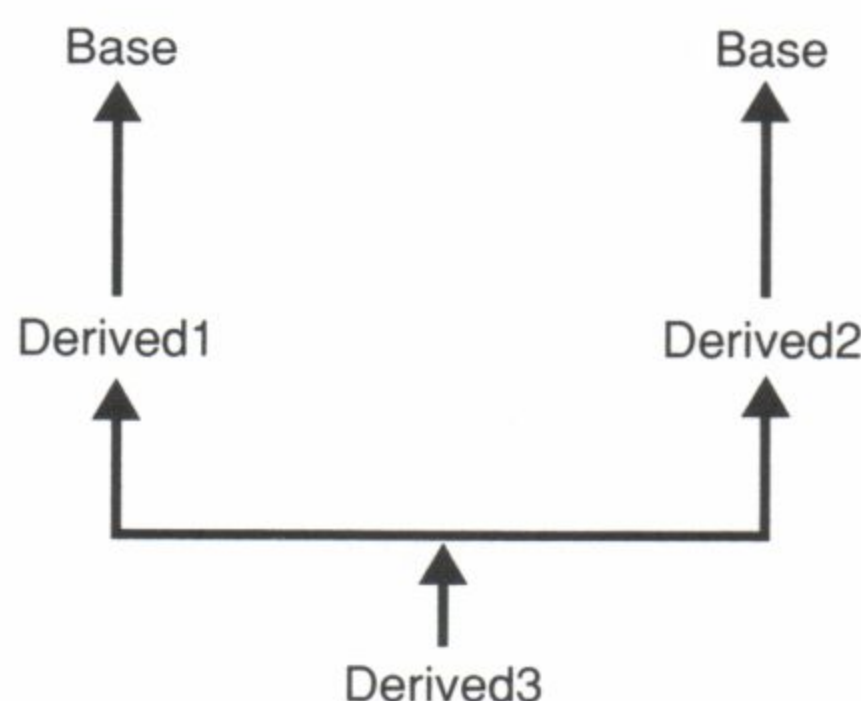
**2.** 次のクラス階層を使用して，Cのコンストラクタを作成しなさい．コンストラクタでは，kを初期化し，A()とB()に引数を渡します．

```
#include <iostream>
using namespace std;

class A {
  int i;
public:
  A(int a) { i = a; }
};

class B {
  int j;
```

```
public:
  B(int a) { j = a; }
};

class C : public A, public B {
  int k;
public:
  /* C()を作成する．kを初期化し，
     A()とB()の両方に引数を渡すこと */
};
```

## 7.5　仮想基本クラス

派生クラスが複数の基本クラスを直接継承すると，それによって問題が引き起こされることがあります．どのような問題かを理解するために，次のクラス階層を考えてみましょう．

Base　　Base
↑　　　↑
Derived1　Derived2
↑　　　↑
└──┬──┘
　　↑
Derived3

基本クラスBaseが，Derived1とDerived2の両方に継承されています．Derived3は，Derived1とDerived2の両方を直接継承しています．しかし，これでは，Derived3がBaseを2回継承することになります．つまり，Derived1を通じて1回，Derived2を通じてもう1回です．これにより，Derived3がBaseのメンバを使用するときに，あいまいさが生じます．Derived3にはBaseのコピーが2つ含まれているので，Baseのメンバへの参照が，Derived1を通じて間接的に継承されたBaseを参照しているのか，Derived2を通じて間接的に継承されたBaseを参照しているのかわかりません．このあいまいさを解決するため，C++には，Derived3にBaseのコピーが1つしか含まれるようにする機構が用意されています．この機能を**仮想基本クラス**（virtual base class）と呼びます．

1つの派生クラスが同じ基本クラスを何度も間接継承するという上記のような場合，すべての派生クラスがその基本クラスをvirtual（仮想）として継承するよう定めておけば，派生オブジェクトの中に同一基本クラスのコピーが2つも含まれる状況を防げます．つまり，その基本クラスを間接継承するどの後

続派生クラスにおいても，その基本クラスのコピーが2つ(またはそれ以上)も存在することはありません．派生クラスに継承される基本クラスでは，基本クラスアクセス指定子の前にvirtualキーワードを指定しておきます．

## 例 7.5 仮想基本クラス

**1.** 次の例では仮想基本クラスを使用し，baseのコピーがderived3に2個も入り込まないようにしています．

```
// このプログラムは仮想基本クラスを使用する
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// 基本クラスをvirtualとして継承する
class derived1 : virtual public base {
public:
  int j;
};

// ここでも，基本クラスをvirtualとして継承する
class derived2 : virtual public base {
public:
  int k;
};

/* derived3は，derived1とderived2の両方を継承しているが，
   基本クラスのコピーは1つしかない */
class derived3 : public derived1, public derived2 {
public:
  int product() { return i * j * k; }
};

int main()
{
  derived3 ob;

  ob.i = 10; // コピーが1つのため，あいまいではない
  ob.j = 3;
  ob.k = 5;
```

```
    cout << "Product is " << ob.product() << '¥n';

    return 0;
  }
```

derived1とderived2がbaseをvirtualとして継承しなかったとすれば，次の文はあいまいです．

```
ob.i = 10;
```

したがって，コンパイルエラーになります(下記練習問題1を参照)．

**2.** 派生クラスによるある基本クラスの継承がvirtualであっても，その基本クラスがその派生クラスの中に存在することは変わりません．この点をよく理解しておいてください．たとえば，上のプログラムにおいて次のプログラムコードは完全に有効です．

```
derived1 ob;
ob.i = 100;
```

通常の基本クラスと仮想基本クラスの違いは，オブジェクトが同じ基本クラスを何度も継承した場合にしか現れません．仮想基本クラスを使用すれば，オブジェクト中にはただ1つの基本クラスしか存在しません．そうでないと，複数のコピーが存在してしまいます．

### 練習問題 7.5 仮想基本クラス

**1.** 例1のプログラムを使い，virtualキーワードを取り除いてコンパイルしなさい．どのようなエラーが生じるか説明しなさい．

**2.** なぜ仮想基本クラスが必要になるのか説明しなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** buildingという基本クラスを作成しなさい．ここには，建物の階数，部屋数，床の総面積を格納します．houseという派生クラスを作成しなさい．これはbuildingを継承し，ほかに寝室数と浴室数を格納します．次にofficeという派生クラスを作成しな

さい．これはbuildingを継承し，ほかに消火器台数と電話台数を格納します．本書巻末の解答と必ずしも一致しなくても，機能的に同じであれば正解と見なします．

**2.** 基本クラスがpublicとして派生クラスに継承されるとき，その公開メンバはどうなるか説明しなさい．また，非公開メンバはどうなるか説明しなさい．基本クラスがprivateとして派生クラスに継承されるとき，その公開メンバと非公開メンバはどうなるか説明しなさい．

**3.** protectedの意味を説明しなさい(クラスのメンバに言及するときの意味と，継承のアクセス指定子として使用するときの意味を，両方とも説明しなさい)．

**4.** あるクラスが別のクラスを継承するとき，その2つのクラスのコンストラクタはどのような順序で呼ばれるか説明しなさい．また，デストラクタはどのような順序で呼ばれるか説明しなさい．

**5.** 次のスケルトンに，コメントに従って詳細を記入しなさい．

```
#include <iostream>
using namespace std;

class planet {
protected:
  double distance; // 太陽からの距離（マイル数）
  int revolve;     // 日数
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // 軌道の円周
public:
  /* earth(double d, int r)を作成する．距離と
     公転日数をplanetに渡し，軌道の円周計算させる
      （ヒント：円周＝2r*3.1416）
  */
  /* この情報を表示するshow()という関数を作成する */
};

int main()
{
  earth ob(93000000, 365);

  ob.show();
```

```
  return 0;
}
```

**6.** 次のプログラムを修正しなさい．

```
/* 車両階層の変形．しかし，このプログラムには
   エラーが含まれているので，それを正すこと．
   ヒント：現在のままコンパイルして，どのような
   エラーメッセージが出るか調べる
*/
#include <iostream>
using namespace std;

// 各種車両の基本クラス
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Wheels: " << num_wheels << '¥n';
    cout << "Range: " << range << '¥n';
  }
};

enum motor {gas, electric, diesel};

class motorized : public vehicle {
  enum motor mtr;
public:
  motorized(enum motor m, int w, int r) : vehicle(w, r)
  {
    mtr = m;
  }
  void showm() {
    cout << "Motor: ";
    switch(mtr) {
      case gas : cout << "Gas¥n";
        break;
      case electric : cout << "Electric¥n";
        break;
      case diesel : cout << "Diesel¥n";
        break;
    }
```

```
  }
};

class road_use : public vehicle {
  int passengers;
public:
  road_use(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void showr()
  {
    cout << "Passengers: " << passengers << '¥n';
  }
};

enum steering { power, rack_pinion, manual };

class car : public motorized, public road_use {
  enum steering strng;
public:
  car(enum steering s, enum motor m, int w, int r, int p) :
    road_use(p, w, r), motorized(m, w, r), vehicle(w, r)
  {
    strng = s;
  }
  void show() {
    showv(); showr(); showm();
    cout << "Steering: ";
    switch(strng) {
      case power : cout << "Power¥n";
        break;
      case rack_pinion : cout << "Rack and Pinion¥n";
        break;
      case manual : cout << "Manual¥n";
        break;
    }
  }
};

int main()
{
  car c(power, gas, 4, 500, 5);

  c.show();

  return 0;
}
```

# 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** 「この章の理解度チェック」の問6で，carとmotorizedに使用されているswitch文について，警告メッセージ(あるいはエラーメッセージ)が出る場合があります．その理由を説明しなさい．

**2.** 前章で学んだとおり，基本クラスでオーバーロードされている演算子のほとんどは，派生クラス内でも使用できます．しかし，使用できないものもあります．それはどの演算子か説明しなさい．なぜ使用できないのかを説明しなさい．

**3.** 次に示すのは，前章で見たcoordクラスの書き直しです．今回は，quadという別のクラスの基本クラスとして使用されています．quadは，ほかに，ある点を含んでいる四分円を保持しています．このプログラムを実行し，出力を理解できるか試してみなさい．

```
/* +, -, =をcoordクラスに関してオーバーロードし,
   そのcoordをquadの基本クラスとして使用する */
#include <iostream>
using namespace std;

class coord {
public:
  int x, y; //  coordinate values
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
  coord operator-(coord ob2);
  coord operator=(coord ob2);
};

// +をcoordクラスに関してオーバーロードする
coord coord::operator+(coord ob2)
{
  coord temp;

  cout << "Using coord operator+()¥n";

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
```

```
  return temp;
}

// -をcoordクラスに関してオーバーロードする
coord coord::operator-(coord ob2)
{
  coord temp;

  cout << "Using coord operator-()¥n";

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// =をcoordクラスに関してオーバーロードする
coord coord::operator=(coord ob2)
{
  cout << "Using coord operator=()¥n";

  x = ob2.x;
  y = ob2.y;

  return *this; // 代入先のオブジェクトを返す
}

class quad : public coord {
  int quadrant;
public:
  quad() { x = 0; y = 0; quadrant = 0; }
  quad(int x, int y) : coord(x, y)
  {
    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;
  }
  void showq()
  {
     cout << "Point in Quadrant: " << quadrant << '¥n';
  }
  quad operator=(coord ob2);
};

quad quad::operator=(coord ob2)
{
  cout << "Using quad operator=()¥n";
```

```
    x = ob2.x;
    y = ob2.y;

    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;

    return *this;
  }

  int main()
  {
    quad o1(10, 10), o2(15, 3), o3;
    int x, y;

    o3 = o1 + o2; // 2つのオブジェクトの加算．演算子+()を呼ぶ
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1+o2) X: " << x << ", Y: " << y << "¥n";

    o3 = o1 - o2; // 2つのオブジェクトの減算
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1-o2) X: " << x << ", Y: " << y << "¥n";

    o3 = o1; // オブジェクトを代入する
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o3=o1) X: " << x << ", Y: " << y << "¥n";

    return 0;
  }
```

**4.** 問3に示すプログラムを適宜変更し，フレンド演算子関数を使用しなさい．

# 8

# C++ の入出力システム

## この章の内容

本書では第1章以降，C++流の入出力を使ってきましたが，ここで少し詳しく見ておくことにしましょう．C++言語には，その前身であるC同様，柔軟かつ強力で，機能豊富な入出力システムが用意されています．C++は，Cの入出力システムも完全にサポートしていますが(この点をよく理解しておいてください)，それに加えてオブジェクト指向の一とおりの入出力ルーチンを完全な形でサポートしています．C++の入出力システムを使用することの利点は，新しく作成するクラスに関してシステムをオーバーロードできることです．つまり，C++の入出力システムでは，開発者が独自に作成する新しい型を違和感なく取り込むことができます．

Cの入出力システムと同様，C++のオブジェクト指向の入出力システムは，コンソール入出力とファイル入出力をほとんど区別しません．ファイル入出力とコンソール入出力の違いは，同一機構を異なる方向から眺めたというだけにすぎません．例ではコンソール入出力を使用しますが，この章で述べる事柄はファイル入出力にも等しく当てはまります(ファイル入出力については，第9章で詳しく取り上げます)．

本書の執筆時点では，使用されている入出力ライブラリに2とおりのバージョンがあります．1つはC++の最初の仕様に基づく古いライブラリ，もう1つは標準C++で定義された新しいライブラリです．この2つのライブラリは，プログラマの目から見てもほとんど違いがありません．新しい入出力ライブラリと言っても，本質的には古いライブラリを更新して改良したものにすぎないからです．両者間に見られる違いの大半は，水面下での出来事です．つまり，実装上の違いであって，使用上の違いではありません．プログラマから見た最大の違いは，新しい入出力ライブラリに新しい機能が2，3追加され，新しいデータ型がいくつか定義されていることです．したがって，新しい入出力ライブラリは，実質的に古い入出力ライブラリのスーパーセットになっています．古いライブラリ用に書かれたプログラムのほとんどは，新しいライブラリでも大きな変更なしでコンパイルできます．こうして，旧入出力ライブラリは新入出力ライブラリで実質的に置き換えられているので，本書では標準C++で定義されている新ライブラリだけを説明します．しかし，説明の大部分は旧ライブラリにも当てはまります．

この章では，書式設定された入出力，入出力マニピュレータ，入出力挿入演算子と入出力抽出演算子の独自作成など，C++の入出力システムのいくつかの側面を取り上げます．C++の入出力システムとCの入出力システムに，共通の機能が多くあることがわかります．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. 航空機についての情報を格納するクラス階層を作成します．まず，airship という基本クラスを作成しなさい．ここに，搭乗可能乗客数と積載可能貨物量(単位はポンド)を格納します．次に，airship から airplane と balloon という 2 つの派生クラスを作成しなさい．airplane には，搭載エンジンの種類(プロペラまたはジェット)と，航続距離(単位はマイル)を格納します．balloon には，気球を浮上させるために使用する気体(水素またはヘリウム)と，最大高度(単位はフィート)を格納します．このクラス階層を使う短いプログラムを作成しなさい(本書の解答と必ずしも一致しなくても，機能的に同じであれば正解と見なします)．

2. protected は，どのような目的に使用されるか説明しなさい．

3. 次のクラス階層が与えられたとき，コンストラクタ関数の呼び出し順序はどうなるか説明しなさい．デストラクタ関数の呼び出し順序はどうなるか説明しなさい．

```
#include <iostream>
using namespace std;

class A {
public:
  A() { cout << "Constructing A¥n"; }
  ~A() { cout << "Destructing A¥n"; }
};

class B : public A {
public:
  B() { cout << "Constructing B¥n"; }
  ~B() { cout << "Destructing B¥n"; }
};

class C : public B {
public:
  C() { cout << "Constructing C¥n"; }
  ~C() { cout << "Destructing C¥n"; }
};

int main()
{
  C ob;
  return 0;
}
```

4. 次のプログラムコードで，コンストラクタ関数の呼び出し順序はどうなるか説明しなさい．

```
class myclass : public A, public B, public C { ...
```

**5.** 次のプログラムに必要なコンストラクタ関数を書き入れなさい.

```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  // コンストラクタが必要
  void showij() { cout << i << ' ' << j << '¥n'; }
};

class derived : public base {
  int k;
public:
  // コンストラクタが必要
  void show() { cout << k << ' '; showij(); }
};

int main()
{
  derived ob(1, 2, 3);

  ob.show();

  return 0;
}
```

**6.** クラス階層を定義するときは，一般に，最も______なクラスから始め，最も______なクラスに至るようにします(抜けている言葉を補いなさい).

# 8.1　C++の入出力の基礎

C++の入出力の説明に入る前に，一般的な事柄を少し述べておきます．C++の入出力システムは，Cの入出力システムと同様，ストリームを通じて動作します．Cでのプログラミング経験のある方はすでにストリームの何たるかをご存じでしょうが，簡単に説明しておきましょう．**ストリーム**(stream)とは，情報を生産もしくは消費する論理デバイスを言います．この論理デバイスが，C++の入出力システムによって物理デバイスとリンクされます．すべてのストリームは，リンク先の物理デバイスが違っても動作そのものは同じです．すべてのストリームの動作が同じなので，この入出力システムはさまざまに能力の異なるデバイスに作用しながら，プログラマに対しては一貫したインターフェイスを提供して

います．たとえば，画面への書き出しに使用する関数を，そのままディスクファイルやプリンタへの書き出しにも使用できます．

ご存じのとおり，C プログラムの実行が始まると，stdin，stdout，stderr という 3 つの定義済みストリームが自動的に開かれます．C++プログラムでも同様のことが起こり，実行が始まると，次の4つのストリームが自動的に開かれます．

**表 8-1　C++ の定義済みストリーム**

| ストリーム | 意味 | デフォルトデバイス |
|---|---|---|
| cin | 標準入力 | キーボード |
| cout | 標準出力 | 画面 |
| cerr | 標準エラー | 画面 |
| clog | cerr のバッファバージョン | 画面 |

cin，cout，cerr の各ストリームが，それぞれ C の stdin，stdout，stderr に対応していることは，ご想像のとおりです．cin と cout はすでに使用してきました．clog ストリームは，バッファを使う cerr にすぎません．標準 C++ は，これらのストリームのワイド(16 ビット)文字バージョンである wcin，wcout，wcerr，wclog も開きますが，本書ではこれらを使用しません．16 ビット文字ストリームは，中国語のような，大きな文字セットを必要とする言語をサポートするためのものです．

標準ストリームは，デフォルトではコンソールとの通信に使用されます．しかし，入出力リダイレクトをサポートしている環境では，これらのストリームをほかのデバイスにリダイレクトできます．

第1章で学んだとおり，C++ は<iostream>というヘッダで入出力システムをサポートします．このヘッダでは，入出力操作をサポートするためのかなり複雑なクラス階層が1組定義されています．入出力クラスは，**テンプレートクラス**(template class)から始まります．テンプレートクラスは**汎用クラス**(generic class)とも呼ばれ，第11章で詳しく取り上げる予定ですが，簡単に言うと，クラスの形式だけを定義するものであり，作用の対象となるデータを完全には指定しません．テンプレートクラスが定義されると，そこから個別のインスタンスを作成できます．入出力ライブラリとの関係で言うと，標準 C++ はどの入出力テンプレートクラスについても 2 とおりのバージョンを作成します．1 つは 8 ビット文字バージョン，もう 1 つはワイド文字バージョンです．本書では，使用頻度がはるかに高い 8 ビット文字クラスだけを扱います．

C++の入出力システムは，2つの(相互に関係はあるものの異なる)テンプレートクラス階層の上に構築されています．1つは，basic_streambuf と呼ばれる低水準入出力クラスから派生したクラス階層で，基本的な低水準入出力操作を実行し，C++の入出力システム全体を下から支えます．特に高度な入出力プログラミングを行うのでない限り，basic_streambuf を直接扱うことはありません．普通に使用するクラス階層は，basic_ios から派生したクラスです．これは高水準入出力クラスで，書式設定とエラー

検査を行い，ストリーム入出力に関係するステータス情報を与えます．basic_iosは，basic_istream，basic_ostream，basic_iostreamなど，いくつかの派生クラスの基本クラスとして使用されます．これらのクラスを使用することによって，basic_istreamは入力，basic_ostreamは出力，basic_iostreamは入出力ストリームの作成に使用されます．

すでに説明したとおり，入出力ライブラリは，ここに挙げたクラス階層を8ビット文字バージョンとワイド文字バージョンの2とおりずつ作成します．次の表に，テンプレートクラス名と，対応する8ビット文字バージョンの名前を示します(第９章で使用するものも含みます)．

表8-2　入出力ライブラリから作成する８ビット文字クラス

| テンプレートクラス | ８ビット文字クラス |
|---|---|
| basic_streambuf | streambuf |
| basic_ios | ios |
| basic_istream | istream |
| basic_ostream | ostream |
| basic_iostream | iostream |
| basic_fstream | fstream |
| basic_ifstream | ifstream |
| basic_ofstream | ofstream |

本書のこれ以後の部分では，8ビット文字クラスの名前を使用します．実際に，プログラムではこの名前が使用されます．また，古い入出力ライブラリでも同じ名前が使われています．旧入出力ライブラリと新入出力ライブラリの間にソースコードレベルでの整合性があるのは，そのためです．

最後にもう一言．iosクラスには，ストリームの基本動作を制御し監視するためのメンバ関数とメンバ変数がいくつも含まれていて，頻繁に参照されます．プログラムに<iostream>をインクルードしておけば，この重要なクラスにアクセスできます．

## 8.2　書式設定された入出力

本書でこれまで示してきた例では，C++のデフォルト書式を使用して，情報を画面に表示してきました．しかし，情報出力にはさまざまな形式を使用できます．Cではprintf()関数を用いてデータの書式を設定しましたが，C++の入出力システムでもほぼ同じ方法で書式設定ができます．また，情報入力の方法も一部変更できます．

どのストリームにも1組の書式フラグが関連付けられていて，情報の書式設定の方法を制御しています．iosクラスでは，fmtflagsという列挙型ビットマスクが宣言されています．ここで定義されている値は，次のとおりです【監注：コンパイラによっては，一部使用できないフラグもあります】．

**表8-3　iosで定義されているfmtflags列挙型のフラグ**

| | | | | | |
|---|---|---|---|---|---|
| adjustfield | dec | hex | oct | showbase | skipws |
| basefield | fixed | internal | right | showpoint | unitbuf |
| boolalpha | floatfield | left | scientific | showpos | uppercase |

これらの値は書式フラグの設定またはクリアに使用され，ios内で定義されています．標準外の古いコンパイラでは，このfmtflags列挙型が定義されていない可能性があります．その場合，書式フラグには長整数型が使用されます．

**skipws**フラグをオンにすると，ストリームの入力時に先行ホワイトスペース文字(スペース，タブ，改行)が破棄されます．skipwsをオフにすると，ホワイトスペース文字は破棄されません．

**left**フラグをオンにすると，出力が左寄せされます．**right**フラグをオンにすると，出力が右寄せされます．**internal**フラグをオンにすると，フィールドがいっぱいになるまで数値の追加が行われます(符号または基数文字の間に必要なだけのスペースが挿入されます)．これらのフラグがどれもオンでないと，出力はデフォルトにより右寄せされます．

数値は，デフォルトでは10進数として出力されますが，基数は変更できます．**oct**フラグをオンにすると，出力は8進数で表示されます．**hex**フラグをオンにすると，16進数で表示されます．出力を10進数に戻すには，**dec**フラグをオンにします．

**showbase**フラグをオンにすると，数値の基数が示されます．たとえば，16進数の値1Fは，0x1Fと表示されます．

デフォルトでは，科学技術表記で表示するときのeは小文字，16進値を表示するときのxは小文字です．**uppercase**フラグをオンにすると，これらの文字が大文字で表示されます．

**showpos**フラグをオンにすると，正値の前に正符号が表示されます．

**showpoint**フラグをオンにすると，すべての浮動小数点出力に小数点と後続ゼロが(必要かどうかにかかわらず)表示されます．

**scientific**フラグをオンにすると，浮動小数点数値が科学技術表記で表示されます．**fixed**をオンにすると，浮動小数点数値が通常表記で表示されます．どちらのフラグもオンにしないと，コンパイラは適切な表記法を選択します．

**unitbuf**フラグをオンにすると，挿入操作のたびにバッファがクリアされます．

**boolalpha**フラグをオンにすると，ブール値の入出力にtrueキーワードとfalseキーワードを使用できます．

oct，dec，hexの各フィールドは頻繁に参照されることから，集合的に**basefield**として参照できます．同様に，left，right，internalの各フィールドは，**adjustfield**として参照できます．scientific，fixedの各フィールドは，**floatfield**として参照できます．

書式フラグを設定するには，setf()関数を使用します．この関数はiosのメンバです．最もよく使われる形式は次のとおりです．

**setf()関数**

```
fmtflags setf(fmtflags flags);
```

この関数は，書式フラグの従来の設定値を返し，flagsに指定されたフラグをオンに設定します(そのほかのフラグは変化しません)．たとえば，showposフラグをオンにするには，次の文を使用できます．

```
stream.setf(ios::showpos);
```

*stream*は，書式の設定を変えたいストリームです．スコープ解決演算子が使われていることに注意してください．showposはiosクラス内の列挙型定数なので，showposの前にクラス名とスコープ解決演算子を置いて，コンパイラにそのことを伝えなければなりません．これを行わないと，定数showposは認識されません．

setf()はiosクラスのメンバ関数であり，そのクラスによって作成されたストリームに作用します．したがって，setf()を呼ぶときは，常に具体的なストリームがあって，それに関して呼ぶことになります．setf()だけを呼ぶということはありえません．言いかえると，C++にはグローバルな書式ステータスという考え方がありません．どのストリームも自分だけの書式ステータス情報を持っています．

1回のsetf()呼び出しで複数のフラグを設定できるため，設定したいフラグの数だけ呼び出しを行う必要はありません．設定したい各フラグの値をORで結んでください．たとえば，次のようにすると，coutのshowbaseフラグとhexフラグがオンに設定されます．

```
cout.setf(ios::showbase | ios::hex);
```

**重要** 書式フラグはiosクラス内で定義されているため，その値にアクセスするには，iosとスコープ解決演算子を使用しなければなりません．たとえば，showbaseだけでは認識されません．ios::showbaseと指定する必要があります．

setf()の反対がunsetf()です．iosのこのメンバ関数は，1つ以上の書式フラグをクリアします．最もよく使われるプロトタイプ形式は，次のとおりです．

unsetf()関数

```
void unsetf(fmtflags flags);
```

*flags* で指定されたフラグがクリアされます(指定されなかったフラグは変化しません).

書式を変えるのでなく，現在どう設定されているかを知りたいだけのときがあります．setf()もunsetf()も1つ以上のフラグの設定を変えるための関数なので，iosにはもう1つ，flags()というメンバ関数が用意されています．これは，各書式フラグの現在の設定状況を返します．そのプロトタイプは，次のとおりです．

flags()関数

```
fmtflags flags( );
```

flags()関数には，もう1つ別の形式があります．こちらを使用すると，あるストリームと関連付けられているすべての書式フラグを，flags()への引数で指定されている値に設定できます．この形式のflags()のプロトタイプは次のとおりです．

flags()関数

```
fmtflags flags(fmtflags f);
```

この形式を使用すると，*f*にあるビットパターンが，そのストリームと関連付けられている書式フラグを保持するための変数にコピーされます．したがって，以前のフラグ設定値はすべて上書きされます．この関数は，以前の設定値を返します．

## 例 8.2　書式設定された入出力

**1.**　次に示すのは，いくつかの書式フラグを設定する例です．

```
#include <iostream>
using namespace std;

int main()
{
  // デフォルト設定で表示する
  cout << 123.23 << " hello " << 100 << '¥n';
  cout << 10 << ' ' << -10 << '¥n';
  cout << 100.0 << "¥n¥n";
```

```
    // 書式を変更する
    cout.unsetf(ios::dec); // コンパイラによっては不要
    cout.setf(ios::hex | ios::scientific);
    cout << 123.23 << " hello " << 100 << '¥n';

    cout.setf(ios::showpos);
    cout << 10 << ' ' << -10 << '¥n';
    cout. setf(ios::showpoint | ios::fixed);
    cout << 100.0;

    return 0;
  }
```

このプログラムの出力は次のようになります.

```
123.23 hello 100
10 -10
100

1.232300e+02 hello 64
a fffffff6
+100.000000
```

showposフラグは，10進出力にしか作用しないことに注意してください．値10を16進数で出力するときは，何も起こりません．また，unsetf()呼び出しでdecフラグをオフにしている(デフォルトではオン)ことに注意してください．この呼び出しは，すべてのコンパイラで必要とされるものではありませんが，コンパイラによっては，decフラグがほかのフラグをオーバーライドすることがあります．その場合は，hexまたはoctをオンにするとき，decをオフにする必要があります．一般に，移植性を高めるためには，使用したい記数法のみを設定し，ほかはオフにしておくのがよいでしょう．

**2.** 次のプログラムでは，uppercaseフラグの効果を確認することができます．最初にuppercase，showbase，hexの各フラグをオンにします．次に88を16進数で出力すると，16進表記に使われるXが大文字になります．次に，unsetf()を使ってuppercaseフラグをクリアし，再び88を16進数で出力すると，今度はxが小文字になります．

```
#include <iostream>
using namespace std;

int main()
{
  cout.unsetf(ios::dec);
  cout.setf(ios::uppercase | ios::showbase | ios::hex);
```

```
  cout << 88 << '¥n';

  cout.unsetf(ios::uppercase);

  cout << 88 << '¥n';

  return 0;
}
```

**3.** 次のプログラムはflags()を使用して，coutの書式フラグの設定値を表示します．特に，showflags関数に注意してください．自分のプログラムで便利に使えるかもしれません．

```
#include <iostream>
using namespace std;

void showflags();
int main()
{
  // 書式フラグのデフォルト状態を示す
  showflags();

  cout.setf(ios::oct | ios::showbase | ios::fixed);

  showflags();

  return 0;
}

// この関数は，書式フラグのステータスを表示する
void showflags()
{
  ios::fmtflags f;

  f = cout.flags();  // フラグの設定値を取得する

  if(f & ios::skipws) cout << "skipws on¥n";
  else cout << "skipws off¥n";

  if(f & ios::left) cout << "left on¥n";
  else cout << "left off¥n";

  if(f & ios::right) cout << "right on¥n";
  else cout << "right off¥n";

  if(f & ios::internal) cout << "internal on¥n";
  else cout << "internal off¥n";
```

```
    if(f & ios::dec) cout << "dec on¥n";
    else cout << "dec off¥n";

    if(f & ios::oct) cout << "oct on¥n";
    else cout << "oct off¥n";

    if(f & ios::hex) cout << "hex on¥n";
    else cout << "hex off¥n";

    if(f & ios::showbase) cout << "showbase on¥n";
    else cout << "showbase off¥n";

    if(f & ios::showpoint) cout << "showpoint on¥n";
    else cout << "showpoint off¥n";

    if(f & ios::showpos) cout << "showpos on¥n";
    else cout << "showpos off¥n";

    if(f & ios::uppercase) cout << "uppercase on¥n";
    else cout << "uppercase off¥n";

    if(f & ios::scientific) cout << "scientific on¥n";
    else cout << "scientific off¥n";

    if(f & ios::fixed) cout << "fixed on¥n";
    else cout << "fixed off¥n";

    if(f & ios::unitbuf) cout << "unitbuf on¥n";
    else cout << "unitbuf off¥n";

    if(f & ios::boolalpha) cout << "boolalpha on¥n";
    else cout << "boolalpha off¥n";

    cout << "¥n";
}
```

showflags()の内部では，ローカル変数fがfmtflags型と宣言されています．ご使用のコンパイラがfmtflagsを定義していないときは，この変数をlongと宣言してください．このプログラムの出力は次のとおりです．

```
skipws on
left off
right off
internal off
dec on
oct off
hex off
showbase off
```

```
showpoint off
showpos off
uppercase off
scientific off
fixed off
unitbuf off
boolalpha off

skipws on
left off
right off
internal off
dec on
oct on
hex off
showbase on
showpoint off
showpos off
uppercase off
scientific off
fixed on
unitbuf off
boolalpha off
```

**4.** 次のプログラムは，もう１つの形式のflags()を使用しています．まず，showpos，showbase，oct，rightをオンにするフラグマスクを作成します．次に，flags()を使用して，coutと関連付けられたフラグ変数をその値に設定します．showflags()関数は，フラグが意図したとおりに設定されていることを確認します(例3のプログラムで使用したものと同じ関数です)．

```
#include <iostream>
using namespace std;

void showflags() ;

int main()
{
  // 書式フラグのデフォルト状態を示す
  showflags();

  // showpos, showbase, oct, rightがオンで, ほかはオフ
  ios::fmtflags f = ios::showpos | ios::showbase |
                    ios::oct | ios::right;

  cout.flags(f);  // フラグを設定

  showflags();
```

```
	return 0;
}
```

## 練習問題　8.2　書式設定された入出力

1. 正の整数の前に+符号を表示するよう，coutのフラグを適切に設定するプログラムを作成しなさい．このプログラムを実行して，書式フラグを正しく設定したことを示しなさい．

2. 浮動小数点値を表示するときに必ず小数点も表示するようにcoutのフラグを適切に設定するプログラムを書きなさい．また，すべての浮動小数点値を科学的記数法で表示し，Eには大文字を使用しなさい．

3. 書式フラグの現在状態を保存し，showbaseとhexをオンに設定し，値100を表示するプログラムを作成しなさい．次に，フラグを以前の値にリセットしなさい．

# 8.3　width()，precision()，fill()の使用

iosでは，書式設定フラグのほかにも，フィールドの幅，精度，充てん文字という3つの書式仮引数があり，それを設定するために，width()，precision()，fill()という3つのメンバ関数が定義されています．

出力される値が占めるスペースは，デフォルトでは，表示に必要な文字数と同数のスペースです．しかし，width()関数を使えば最小フィールド幅を指定できます．width()関数のプロトタイプは次のとおりです．

width()関数

```
streamsize width(streamsize w);
```

*w*はフィールド幅です．従来のフィールド幅が返されます．streamsize型は，<iostream>で何らかの整数として定義されています．出力操作のたびにフィールド幅をデフォルト設定値に戻す実装方法もあって，その場合は，出力文を書くたびに，事前に最小フィールド幅を設定しなければなりません．

最小フィールド幅が設定されていて，実際の値が指定された最小フィールド幅に満たないと，そのフィールドは，フィールド幅いっぱいまで現在の充てん文字(デフォルトではスペース)で埋められま

す．反対に，出力値のサイズが最小フィールド幅を超えると，フィールドの桁あふれが発生します．値の一部が切り捨てられることはありません．

精度は，デフォルトでは6桁ですが，precision()関数で自由に設定できます．プロトタイプは次のとおりです．

precision()関数

```
streamsize precision(streamsize p);
```

精度は*p*に設定されます．従来の精度が返されます．

フィールドを満たす必要がある場合，デフォルトではスペースが使われますが，fill()関数で任意の充てん文字を指定できます．fill()関数のプロトタイプは次のとおりです．

fill()関数

```
char fill(char ch);
```

このfill()が呼ばれると，*ch*が新しい充てん文字になり，古い充てん文字が返されます．

## 例 8.3 width()，precision()，fill()の使用

**1.** 次に，書式関数の使い方を示すプログラムを示します．

```
#include <iostream>
using namespace std;

int main()
{
  cout.width(10);            // 最小フィールド幅を設定する
  cout << "hello" << '¥n'; // デフォルトにより右寄せ
  cout.fill('%');            // 充てん文字を設定する
  cout.width(10);            // 幅を設定する
  cout << "hello" << '¥n'; // デフォルトにより右寄せ
  cout.setf(ios::left);      // 左寄せ
  cout.width(10);            // 幅を設定する
  cout << "hello" << '¥n'; // 左寄せで出力する

  cout.width(10);            // 幅を設定する
  cout.precision(10);        // 精度を10桁に設定する
  cout << 123.234567 << '¥n';
  cout.width(10);            // 幅を設定する
  cout.precision(6);         // 精度を6桁に設定する
```

```
  cout << 123.234567 << '¥n';
  return 0;
}
```

このプログラムの出力は次のようになります.

```
    hello
%%%%%hello
hello%%%%%
123.234567
123.235%%%
```

フィールド幅が，どの出力文の前でも設定されていることに注意してください.

**2.** 次のプログラムでは，C++の入出力書式関数を使って数表を作成します.

```
// 平方根と平方の表を作成する
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double x;

  cout.precision(4);
  cout << "      x   sqrt(x)     x^2¥n¥n";

  for(x = 2.0; x <= 20.0; x++) {
    cout.width(7);
    cout << x << "   ";
    cout.width(7);
    cout << sqrt(x) << "   ";
    cout.width(7);
    cout << x*x << '¥n';
  }

  return 0;
}
```

このプログラムは，次の表を作成します.

```
      x   sqrt(x)     x^2

      2     1.414       4
      3     1.732       9
      4         2      16
      5     2.236      25
      6     2.449      36
```

```
   7     2.646       49
   8     2.828       64
   9         3       81
  10     3.162      100
  11     3.317      121
  12     3.464      144
  13     3.606      169
  14     3.742      196
  15     3.873      225
  16         4      256
  17     4.123      289
  18     4.243      324
  19     4.359      361
  20     4.472      400
```

**練習問題　8.3　width()，precision()，fill()の使用**

1. 2から100までの数値について，自然対数と常用対数を表示するプログラムを作成しなさい．数値は，幅10桁のフィールドに右寄せし，小数点以下5桁まで表示します．

2. 次のようなプロトタイプを持つcenter()という関数を作成しなさい．

   ```
   void center(char *s);
   ```

   この関数は，指定された文字列を画面の中央に表示します．width()関数を使用してください．画面は80文字幅とします(話を簡単にするため，文字列は80文字を超えないものとします)．プログラムを作成して，この関数の働きを試しなさい．

3. 書式フラグと書式関数をいろいろと試してみなさい．C++の入出力システムに慣れれば，これを使って出力をどのような書式にでも設定できます．

# 8.4　入出力マニピュレータの使用

C++の入出力システムを使って情報の書式を設定するには，もう1つの方法があります．こちらの方法では，**入出力マニピュレータ**(I/O manipulator)という特殊な関数を使用します．先に進むにつれて，場合によっては，入出力マニピュレータの方がiosの書式フラグや書式関数より使いやすいことがわかってくると思います．

入出力マニピュレータは，入出力文の内部で使用できる特別な入出力書式関数です．入出力文と切り離さなければならなかったiosメンバ関数とは，その点で異なります．標準のマニピュレータを表8-1

に示しておきます．ご覧のとおり，iosクラスのメンバ関数と対応している入出力マニピュレータが少なくありません．表8-4に示すマニピュレータの多くは，最近になって標準C++に追加されましたから，最新のコンパイラでしかサポートされていません．

setw()のように仮引数を取るマニピュレータもあって，これにアクセスするには，プログラムに<iomanip>をインクルードしておかなければなりません．引数を要求しないマニピュレータでは，その必要はありません．

**表8-4　標準C++の入出力マニピュレータ**

| マニピュレータ | 目的 | 入出力 |
|---|---|---|
| boolalpha | boolalphaフラグをオン | 入出力 |
| dec | decフラグをオン | 入出力 |
| endl | 改行文字(¥n)を出力し，ストリームをフラッシュする | 出力 |
| ends | ヌルを出力 | 出力 |
| fixed | fixedフラグをオン | 出力 |
| flush | ストリームをクリア | 出力 |
| hex | hexフラグをオン | 入出力 |
| internal | internalフラグをオン | 出力 |
| left | leftフラグをオン | 出力 |
| noboolalpha | boolalphaフラグをオフ | 入出力 |
| noshowbase | showbaseフラグをオフ | 出力 |
| noshowpoint | showpointフラグをオフ | 出力 |
| noshowpos | showposフラグをオフ | 出力 |
| noskipws | skipwsフラグをオフ | 入力 |
| nounitbuf | unitbufフラグをオフ | 出力 |
| nouppercase | uppercaseフラグをオフ | 出力 |
| oct | octフラグをオン | 入出力 |
| resetiosflags(fmtflags *f*) | fに指定されているフラグをオフ | 入出力 |
| right | rightフラグをオン | 出力 |
| scientific | scientificフラグをオン | 出力 |
| setbase(int *base*) | 基数をbaseに設定 | 入出力 |
| setfill(int *ch*) | 充てん文字をchに設定 | 出力 |
| setiosflags(fmtflags *f*) | fで指定されたフラグをオン | 入出力 |
| setprecision(int *p*) | 精度の桁数を設定 | 出力 |
| setw(int *w*) | フィールド幅をwに設定 | 出力 |
| showbase | showbaseフラグをオン | 出力 |
| showpoint | showpointフラグをオン | 出力 |
| showpos | showposフラグをオン | 出力 |

| マニピュレータ | 目的 | 入出力 |
|---|---|---|
| skipws | skipws フラグをオン | 入力 |
| unitbuf | unitbuf フラグをオン | 出力 |
| uppercase | uppercase フラグをオン | 出力 |
| ws | 先行ホワイトスペースをスキップ | 入力 |

前述のとおり，マニピュレータは一連の入出力操作の中で使用できます．次に例を示します．

```
cout << oct << 100 << hex << 100;
cout << setw(10) << 100;
```

最初の文は，まず整数を8進数で表示するようcoutに指示して，100を8進数で出力し，次に整数を16進数で表示するように指示して，100を16進数で出力しています．2つ目の文はフィールド幅を10桁に設定し，再び100を16進数で表示します．マニピュレータが引数を取らないとき(たとえば，oct)は，後ろに括弧が付かないことに注意してください．これは，オーバーロードされた<<演算子に渡されるのがマニピュレータのアドレスであるためです．

入出力マニピュレータは，当該入出力式を含んでいるストリームにしか作用しないことを覚えておいてください．現在開かれていて使用できる，すべてのストリームに作用するのではありません．

上の例でもわかるとおり，iosメンバ関数と比べたときのマニピュレータの利点は，使いやすいことと，コンパクトなコードが書けることです．

マニピュレータを使い，個々の書式フラグを手作業で設定したいときは，setiosflags()を使います．このマニピュレータは，setf()メンバ関数と同じ働きをします．フラグをオフにするには，resetiosflags()マニピュレータを使用します．このマニピュレータは，unsetf()と同じ働きをします．

## 例 8.4　入出力マニピュレータの使用

**1.** 次のプログラムでは，いくつかの入出力マニピュレータを使っています．

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << hex << 100 << endl;
  cout << oct << 10 << endl;

  cout << setfill('X') << setw(10);
  cout << 100 << " hi " << endl;
```

```
  return 0;
}
```

このプログラムの出力は次のようになります.

```
64
12
XXXXXXX144 hi
```

**2.** 先ほど，2から20までの整数の平方と平方根を表形式で表示するプログラムを見ました．次に示すのは，これを書き直したプログラムです．このバージョンでは，メンバ関数と書式フラグの代わりに入出力マニピュレータを使います．

```
/* このバージョンは，入出力マニピュレータを使って，
   平方と平方根の表を表示する */
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x;

  cout << setprecision(4);
  cout << "      x   sqrt(x)     x^2¥n¥n";

  for(x = 2.0; x <= 20.0; x++) {
    cout << setw(7) << x << "  ";
    cout << setw(7) << sqrt(x) << "  ";
    cout << setw(7) << x*x << '¥n';
  }

  return 0;
}
```

**3.** 新しい入出力ライブラリで追加された書式フラグのうち，特に興味深いものの1つにboolalphaがあります．このフラグは，直接設定することも，新しいマニピュレータboolalphaまたはnoboolalphaを使って設定することもできます．boolalphaの面白い点は，これをオンに設定しておくことにより，ブール値の入出力にtrueとfalseというキーワードを使用できることです．通常は，真には1，偽には0を入力しなければなりません．たとえば，次のプログラムを見てください．

```
// boolalpha書式フラグの使用例
#include <iostream>
using namespace std;
```

```
int main()
{
  bool b;

  cout << "Before setting boolalpha flag: ";
  b = true;
  cout << b << " ";
  b = false;
  cout << b << endl;

  cout << "After setting boolalpha flag: ";
  b = true;
  cout << boolalpha << b << " ";
  b = false;
  cout << b << endl;

  cout << "Enter a Boolean value: ";
  cin >> boolalpha >> b; // trueまたはfalseを入力できる
  cout << "You entered " << b;

  return 0;
}
```

このプログラムの出力は次のようになります．

```
Before setting boolalpha flag: 1 0
After setting boolalpha flag: true false
Enter a Boolean value: true
You entered true
```

このように，boolalpha フラグを設定しておくと，ブール値の入出力に true と false という単語が使用されるようになります．boolalpha フラグは，cin と cout で別個に設定しなければならないことに注意してください．ほかの書式フラグ同様，あるストリームに boolalpha を設定しても，別のストリームには適用されません．

## 練習問題　8.4　入出力マニピュレータの使用

**1.** 8.3 節の練習問題 1 と 2 を，メンバ関数と書式フラグの代わりに入出力マニピュレータを使って書き直しなさい．

**2.** 値 100 を 16 進数で出力し，プレフィックス (0x) も表示する入出力文は，どうなるか説明しなさい．setiosflags() マニピュレータを使用してください．

**3.** boolalpha フラグを設定することの効果を説明しなさい．

# 8.5　独自挿入子の作成

すでに説明したとおり，C++の入出力システムを使用することの利点の1つは，作成したクラスに合わせて入出力演算子をオーバーロードできることです．これにより，作成したクラスをC++プログラムにシームレスに取り入れることができます．次に，C++の出力演算子<<をオーバーロードする方法を学びます．

C++の用語では，出力操作を**挿入**(insertion)と呼び，<<を**挿入演算子**(insertion operator)と呼びます．<<を出力用にオーバーロードするということは，**挿入関数**(insertion function)，または**挿入子**(inserter)を作成するということです．この用語の使い方は，出力演算子が情報をストリームに挿入することに由来しています．

挿入関数の一般形式は次のとおりです．

**挿入関数**

```
ostream &operator <<(ostream &stream, class-name ob)
{
    // 挿入子の本体
    return stream;
}
```

最初の仮引数は，ostream型オブジェクトへの参照です．これは，*stream*が出力ストリームでなければならないことを意味しています(*ostream*は，iosクラスから派生しています)．2番目の仮引数は，出力されるオブジェクトを受け取ります(アプリケーションによっては，これが参照仮引数であってもかまいません)．挿入関数が*stream*への参照を返すことと，それがostream型であることに注意してください．この2点は，オーバーロードされた<<を，次のような一連の入出力式で使うのに，必須な条件です．

```
cout << ob1 << ob2 << ob3;
```

挿入子では，いかなる種類の手続きも実行できます．挿入子に何をさせるかは，プログラマ次第です．しかし，良いプログラミング習慣と言える範囲内で挿入子を使おうとするなら，その働きをストリームへの情報出力だけにとどめるべきでしょう．

最初は意外に思われるかもしれませんが，挿入子は，作用対象のクラスのメンバにすることができません．なぜでしょうか．どのような種類の演算子関数であっても，それがあるクラスのメンバになっているときは，thisポインタによって暗黙的に引き渡される左オペランドが，その演算子関数への呼び出しを生成するオブジェクトとなります．当然，この左オペランドは当該クラスのオブジェクトということになります．つまり，オーバーロードされた演算子関数が何らかのクラスのメンバであるときは，左

オペランドも同じクラスのオブジェクトでなければなりません．しかし，挿入子を作成するときは，左オペランドがストリーム，右オペランドが出力したいオブジェクトです．したがって，挿入子はメンバ関数ではありえません．

挿入子がメンバ関数になれないという事実は，C++の重大な欠陥のように見えるかもしれません．なぜなら，挿入子で出力されるクラスのデータはすべて公開でなければならないように見え，それはカプセル化の大原則に違反しているように思われるからです．しかし，それは違います．挿入子は，作用対象であるクラスのメンバにはなれませんが，そのクラスのフレンドにはなることができます．実際，現実に発生するほとんどのプログラミング状況では，オーバーロードされた挿入子は，作用対象となるクラスのフレンドになります．

## 例 8.5　独自挿入子の作成

**1.** まず，簡単な例を見ておきましょう．次のプログラムは，第6章，第7章で開発した coord クラス用の挿入子を含んでいます．

```
// coord型オブジェクトにフレンド挿入子を使用する
#include <iostream>
using namespace std;

class coord {
  int x, y;
public:
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
  friend ostream &operator<<(ostream &stream, coord ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '¥n';
  return stream;
}

int main()
{
  coord a(1, 1), b(10, 23);
  cout << a << b;

  return 0;
}
```

このプログラムの出力は次のようになります．

```
1, 1
10, 23
```

このプログラムには，独自に挿入子を作成する際の非常に重要なポイントが示されています．それは，できる限り汎用的にするということです．このプログラムで見ると，挿入子内の入出力文はxとyの値をstreamに出力します．そのstreamとは，たまたま関数に渡されるどのようなストリームでもかまいません．次章で見るとおり，画面への出力を行う挿入子は，正しく書いておきさえすればどのようなストリームへの出力にも使用できます．初心者は，coord挿入子をつい次のように書いてしまうことがあります．

```
{
  cout << ob.x << ", " << ob.y << '¥n';
  return stream;
}
```

この出力文は，coutにリンクしている標準出力デバイスに情報を表示するようにハードコードされています．これでは，この挿入子をほかのストリームに使用することができません．要するに，挿入子はできる限り汎用的に作成するべきです．そうすることで生じる不都合は何もありません．

**2.** 参考までに上のプログラムを手直しし，挿入子をcoordクラスのフレンドでなくしてみました．この挿入子はcoordの非公開部分にアクセスできないため，変数xとyを公開にしなければなりません．

```
/* coord型のオブジェクト用に挿入子を作成する.
   非フレンド挿入子を使用 */
#include <iostream>
using namespace std;

class coord {
public:
  int x, y;  // 公開でなければならない
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
};

// coordクラス用の挿入子
ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '¥n';
  return stream;
}
```

```
int main()
{
  coord a(1, 1), b(10, 23);
  cout << a << b;

  return 0;
}
```

**3.** 挿入子の働きは，テキスト情報の表示にとどまりません．特定デバイスまたは特定状況の要求する形式で情報を出力することを目的に，どのような操作や変換でも実行できます．たとえば，情報をプロッタに送る挿入子を作成することも難なくできます．この場合，挿入子は，情報のほかに必要なプロッタコードを送らなければなりません．この種の挿入子の感触をつかんでいただくため，次のプログラムを用意しました．このプログラムは，triangleというクラスを作成します．ここには，直角三角形の幅と高さが格納されます．このクラスの挿入子は，画面に三角形を表示します．

```
// このプログラムは，直角三角形を描く
#include <iostream>
using namespace std;

class triangle {
  int height, base;
public:
  triangle(int h, int b) { height = h; base = b; }
  friend ostream &operator<<(ostream &stream, triangle ob);
};

// 三角形を描く
ostream &operator<<(ostream &stream, triangle ob)
{
  int i, j, h, k;

  i = j = ob.base-1;
  for(h=ob.height-1; h; h--) {
    for(k=i; k; k--)
      stream << ' ';
    stream << '*';

    if(j!=i) {
      for(k=j-i-1; k; k--)
        stream << ' ';
      stream << '*';
    }

    i--;
    stream << '¥n';
```

```
  }
  for(k=0; k<ob.base; k++) stream << '*';
  stream << '¥n';

  return stream;
}

int main()
{
  triangle t1(5, 5), t2(10, 10), t3(12, 12);

  cout << t1;
  cout << endl << t2 << endl << t3;

  return 0;
}
```

正しく設計された挿入子であれば，「通常」の入出力式に完全に統合できることが，このプログラムからもわかります．このプログラムは，次のように表示します．

```
    *
   **
  * *
 *  *
*****

         *
        **
       * *
      *  *
     *   *
    *    *
   *     *
  *      *
 *       *
**********

           *
          **
         * *
        *  *
       *   *
      *    *
     *     *
    *      *
   *       *
  *        *
 *         *
************
```

## 練習問題　8.5　独自挿入子の作成

**1.** 次の strtype クラスとプログラムの一部が与えられたとき，文字列を表示する挿入子を作成しなさい．

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype() {delete [] p;}
  friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr)+1;
  p = new char [len];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

// 演算子<<の挿入関数をここに作成する

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  cout << s1 << '¥n' << s2;

  return 0;
}
```

**2.** 次のプログラムの show()関数を挿入関数で置き換えなさい．

```
#include <iostream>
using namespace std;

class planet {
```

```
protected:
  double distance; // 太陽からの距離（マイル数）
  int revolve;     // 日数
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // 軌道の円周
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }

 /* これを書き直し，情報表示に
    挿入関数を使う */
  void show() {
    cout << "Distance from sun: " << distance << '¥n';
    cout << "Days in obit: " << revolve << '¥n';
    cout << "Circumference of orbit: " << circumference
    ➡<< '¥n';
  }
};

int main()
{
  earth ob(93000000, 365);

  cout << ob;

  return 0;
}
```

**3.** なぜ挿入子はメンバ関数になりえないのか説明しなさい．

## 8.6　抽出子の作成

<< 出力演算子をオーバーロードできるのと同様に，>> 入力演算子もオーバーロードできます．C++では，>> を**抽出演算子**(extraction operator）と呼び，それをオーバーロードする関数を**抽出子**(extractor)と呼んでいます．抽出というのは，ストリームから情報を入力する作業が，ストリームからのデータ取り出し(抽出)であることによります．

抽出関数の一般形式は次のとおりです．

**抽出関数**

```
istream &operator>>(istream &stream, class-name &ob)
{
    // 抽出子の本体
    return stream;
}
```

抽出子はistreamへの参照を返します．これは入力ストリームです．最初の仮引数は，入力ストリームへの参照でなければなりません．2番目の仮引数は，入力を受け取るオブジェクトへの参照です．

挿入子がメンバ関数になりえないのと同じ理由で，抽出子はメンバ関数であることができません．抽出子ではどのような操作も実行できますが，情報入力に限定しておいた方がよいでしょう．

## 例 8.6　抽出子の作成

**1.** 次のプログラムは，coord クラスに抽出子を付け加えています．

```
// coord型オブジェクトにフレンド抽出子を追加する
#include <iostream>
using namespace std;

class coord {
  int x, y;
public:
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
  friend ostream &operator<<(ostream &stream, coord ob);
  friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '¥n';
  return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
  cout << "Enter coordinates: ";
  stream >> ob.x >> ob.y;
  return stream;
}

int main()
```

```
{
  coord a(1, 1), b(10, 23);

  cout << a << b;
  cin >> a;
  cout << a;

  return 0;
}
```

この抽出子はユーザーにプロンプトを出力し，入力を要求していることに注意してください．ほとんどの状況では，このようなプロンプトは必要ありません(望ましくもありません)が，カスタマイズした抽出子を使えば，プロンプトメッセージを出力するためのコーディングが簡単になることが上の例からわかります．

**2.** 次のプログラムはinventoryクラスを作成しています．このクラスには，品名，在庫数量，原価を格納します．また，このクラスの挿入子と抽出子を作成しています．

```
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
  char item[40]; // 品名
  int onhand;    // 在庫数量
  double cost;   // 原価
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " on hand at $" << ob.cost << '¥n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
```

```
  cout << "Enter item name: ";
  stream >> ob.item;
  cout << "Enter number on hand: ";
  stream >> ob.onhand;
  cout << "Enter cost: ";
  stream >> ob.cost;

  return stream;
}

int main()
{
  inventory ob("hammer", 4, 12.55);

  cout << ob;
  cin >> ob;
  cout << ob;

  return 0;
}
```

### 練習問題 8.6 抽出子の作成

1. 8.5 節の練習問題 1 の strtype クラスに抽出子を追加しなさい．

2. 整数値とその最小因数を格納するクラスを作成しなさい．このクラスの挿入子と抽出子を作成しなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. 値 100 を，10 進数，16 進数，8 進数で表示するプログラムを書きなさい (ios 書式フラグを使用します)．

2. 値 1000.5364 を 20 文字フィールドに左寄せし，小数点以下 2 桁まで表示して，余白を充てん文字 * で埋めるプログラムを書きなさい (ios 書式フラグを使用します)．

3. 練習問題 1，2 の解答を，入出力マニピュレータを使って書き直しなさい．

**4.** coutの書式フラグを保存し，復元する方法を示しなさい．メンバ関数を使用する方法，マニピュレータを使用する方法のいずれでもかまいません．

**5.** 次のクラスの挿入子と抽出子を作成しなさい．

```
class pwr {
  int base;
  int exponent;
  double result; // baseのexponent乗
public:
  pwr(int b, int e);
};

pwr::pwr(int b, int e)
{
  base = b;
  exponent = e;

  result = 1;
  for( ; e; e--) result = result * base;
}
```

**6.** 正方形の寸法を格納するboxというクラスを作成しなさい．画面に正方形を表示する挿入子を作成しなさい(正方形の表示は好きな方法で行ってください)．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** 次に示すstackクラスを使用して，スタックの内容を表示する挿入子を作成しなさい．その挿入子が正しく機能するかどうか，実行してみてください．

```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字型stackクラスを宣言する
class stack {
  char stck[SIZE]; // スタックを保持する
  int tos;         // スタックの最上部のインデックス
```

```
public:
  stack();
  void push(char ch); // 文字をスタックにプッシュする
  char pop();         // 文字がスタックからポップする
};

// スタックを初期化する
stack::stack()
{
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字がポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0; // スタックが空ならnullを返す
  }
  tos--;
  return stck[tos];
}
```

2. watchというクラスを含むプログラムを作成しなさい．標準時刻関数を使用し，クラスのコンストラクタにシステム時刻を読ませて，格納させます．その時刻を表示する挿入子を作成しなさい．

3. 次に示すクラスは，フィートをインチに変換します．これを使用して，ユーザーにフィート値入力を求める抽出子を作成しなさい．また，フィート数とインチ数を表示する挿入子を作成しなさい．また，その挿入子と抽出子が正しく動作するかどうかを試すプログラムを作成しなさい．

```
class ft_to_inches {
  double feet;
  double inches;
```

```
public:
  void set(double f) {
    feet = f;
    inches = f * 12;
  }
};
```

# 9

# C++ の高度な入出力システム

## この章の内容

この章では，引き続きC++の入出力システムについて学習します．まず，独自の入出力マニピュレータを作成し，それによってファイルを操作する方法を学びます．C++の入出力システムは内容が豊富で柔軟性が高く，数多くの機能を含んでいます．本書でその機能をすべて紹介することはできないので，ここでは最も重要なものだけいくつか取り上げることにします．

メモ　この章で説明するC++の入出力システムは，標準C++の定義に基づいているので，主要なすべてのC++コンパイラと互換性があります．古いコンパイラや，標準に準拠していないコンパイラの入出力システムは，ここで説明する機能をすべて持っているとは限りません．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

**1.** 「`C++ is fun`」という文を40文字幅フィールドに表示するプログラムを作成しなさい．充てん文字にはコロン(:)を使用します．

**2.** 10/3の結果を小数点以下3桁まで表示するプログラムを作成しなさい．iosメンバ関数を使用します．

**3.** 問2のプログラムを，入出力マニピュレータを使って書き直しなさい．

**4.** 挿入子とは何か，抽出子とは何か，それぞれ説明しなさい．

**5.** 次のクラスの挿入子と抽出子を作成しなさい．

```
class date {
  char d[9]; // 日付をmm/dd/yyという文字列として格納する
public:
  // 挿入子と抽出子を付け加える
};
```

**6.** プログラムで，仮引数を取る入出力マニピュレータを使用するとき，どのようなヘッダをインクルードしなければならないか説明しなさい．

**7.** C++プログラムの実行が始まるとき，どの定義済みストリームが作成されるか述べなさい．

# 9.1 独自マニピュレータの作成

C++の入出力システムをカスタマイズする方法としては，挿入演算子と抽出演算子のオーバーロードに加えて，独自のマニピュレータ関数を作成する方法があります．カスタムマニピュレータは，主として次の2つの理由から重要です．まず，マニピュレータを使えば，それぞれ独立した一連の入出力操作を1つにまとめることができます．1つのプログラムの中では，同じ入出力操作の流れが何度も繰り返されることが珍しくありません．そのような場合，カスタムマニピュレータを使って実行すれば，ソースコードが単純になり，書き間違いがなくなります．次に，標準外のデバイスで入出力操作を実行する必要があるときは，カスタムマニピュレータが重要になることがあります．たとえば，特殊なプリンタや光学式文字認識システム(OCR)に制御コードを送る場合に，マニピュレータを使用できます．

カスタムマニピュレータは，オブジェクト指向プログラミングをサポートするC++の機能ですが，オブジェクト指向でないプログラムでも役に立ちます．いずれ説明するとおり，カスタムマニピュレータを使うと，入出力の多いプログラムがわかりやすく，効率的になります．

すでにご存じのとおり，マニピュレータには入力ストリームに作用するマニピュレータと出力ストリームに作用するマニピュレータの2種類があります．このほか，引数を取るマニピュレータと，取らないマニピュレータという分類もあります．仮引数の有無で，マニピュレータの作り方がかなり違ってきます．仮引数を取るマニピュレータの作成は，そうでないマニピュレータの作成よりかなり難しく，本書の範囲を超えます．しかし，仮引数を取らないマニピュレータを独自に作成するのはきわめて簡単です．それを次に説明します．

仮引数を取らない出力マニピュレータ関数のスケルトンは，次のとおりです．

**出力マニピュレータ関数**

```
ostream &manip-name(ostream &stream)
{
   // ここに独自のプログラムコードを書く
   return stream;
}
```

*manip-name*はマニピュレータの名前です．*stream*は，呼び出しを行うストリームへの参照です．そのストリームへの参照が返されます．これが返されないと，マニピュレータをより大きな入出力式の一部として使用することができません．マニピュレータは，作用対象のストリームへの参照を唯一の引数として受け取りますが，出力操作の中で呼び出されるときは引数を使用しません．この点を理解しておくことが重要です．

仮引数を取らない入力マニピュレータ関数のスケルトンは，次のとおりです．

**入力マニピュレータ関数**

```
istream &manip-name(istream &stream)
{
  // ここに独自のプログラムコードを書く
  return stream;
}
```

入力マニピュレータは，呼び出しを行ったストリームへの参照を受け取ります．マニピュレータはこのストリームを返さなければなりません．

**重要** マニピュレータは，呼び出しを行ったストリームへの参照を返します．これを行わないと，一連の入力操作または出力操作の流れの中でマニピュレータを使用することができません．

## 例 9.1　独自マニピュレータの作成

**1.** 最初に簡単な例を見ておきましょう．次のプログラムは，setup() というマニピュレータを作成します．setup() はフィールド幅を 10 桁，精度を 4 桁，充てん文字を * に設定します．

```
#include <iostream>
using namespace std;

ostream &setup(ostream &stream)
{
  stream.width(10);
  stream.precision(4);
  stream.fill('*');

  return stream;
}

int main()
{
  cout << setup << 123.123456;

  return 0;
}
```

ご覧のとおり，setup() は組み込みマニピュレータとまったく同じ方法で，入出力式の一部として使用されます．

**2.** カスタムマニピュレータは，特に複雑なものでなくても十分に役に立ちます．たとえば，次に示すマニピュレータ atn() と note() はごく簡単なものですが，頻繁に使用される語句を簡単に出力できるようにします．

```
#include <iostream>
using namespace std;

// 警告
ostream &atn(ostream &stream)
{
  stream << "Attention: ";

  return stream;
}

// 注意書き
ostream &note(ostream &stream)
{
  stream << "Please Note: ";

  return stream;
}

int main()
{
  cout << atn << "High voltage circuit¥n";
  cout << note << "Turn off all lights¥n";

  return 0;
}
```

簡単なマニピュレータですが，使用頻度の高さを考えれば，タイプ作業を大幅に減らす便利な手段となります．

**3.** 次のプログラムは，getpass() という入力マニピュレータを作成しています．このマニピュレータはビープ音を鳴らし，パスワードの入力を要求します．

```
#include <iostream>
#include <cstring>
using namespace std;

// 簡単な入力マニピュレータ
istream &getpass(istream &stream)
{
  cout << '¥a';  // ビープ音を鳴らす
  cout << "Enter password: ";
```

```
  return stream;
}

int main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "password"));

  cout << "Logon complete¥n";

  return 0;
}
```

**練習問題　9.1　独自マニピュレータの作成**

1. 現在のシステム時刻とシステム日付を表示する出力マニピュレータを作成しなさい．マニピュレータの名前をtd()とします．

2. sethex()という出力マニピュレータを作成しなさい．sethex()は出力を16進に設定し，uppercaseフラグとshowbaseフラグをオンにします．また，reset()という出力マニピュレータを作成しなさい．これは，sethex()が行った変更を元に戻します．

3. skipchar()という入力マニピュレータを作成しなさい．skipchar()は入力ストリームから次の10文字を読み取って，無視します．

## 9.2　ファイル入出力の基本

次に，ファイル入出力の説明に移りましょう．前章で述べたとおり，ファイル入出力とコンソール入出力は密接に関連しています．事実，コンソール入出力をサポートするクラス階層が，そのままファイル入出力もサポートしています．したがって，これまで入出力について学んできたことは，その大部分がファイルにも当てはまります．もちろん，ファイル処理には，いくつかの新しい機能も使用されます．

ファイル入出力を実行するには，プログラムに<fstream>ヘッダをインクルードしておかなければなりません．このヘッダでは，ifstream，ofstream，fstreamなど，いくつかのクラスが定義されています．どのクラスもistreamとostreamから派生していて，そのistreamとostreamはiosから派生

しているので，ifstream，ofstream，fstreamでは，iosで定義されているすべての操作にアクセスできます(iosについては，前章で説明しました).

C++では，ストリームにリンクさせることでファイルを開きます．ストリームには，入力，出力，入出力の3種類があります．ファイルを開くためには，まずストリームを取得しなければなりません．入力ストリームを作成するには，ifstream型のオブジェクトを宣言します．出力ストリームを作成するには，ofstream型のオブジェクトを宣言します．入力操作と出力操作の両方を実行するストリームは，fstream型のオブジェクトとして宣言しなければなりません．たとえば，次のプログラムコードは，入力ストリームを1つ，出力ストリームを1つ，入出力がともに可能なストリームを1つ作成します．

```
ifstream in;   // 入力
ofstream out;  // 出力
fstream io;    // 入出力
```

作成したストリームをファイルと関連付けるには，open()を使用するのが1つの方法です．この関数は，3つのファイルストリームクラスすべてのメンバです．それぞれのopen()関数のプロトタイプを次に示します．

open()関数

```
void ifstream::open(const char *filename,
                    openmode mode = ios::in);
void ofstream::open(const char *filename,
                    openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename,
                   openmode mode = ios::in | ios::out);
```

*filename*はファイルの名前です．パス指定子を含んでいてもかまいません．*mode*は，ファイルの開く方法を指定します．これはopenmode型の値でなければなりません．openmode型はiosで定義されている列挙型で，次の値を含みます．

**表9-1　ファイルを開く方法を指定するmodeの値**

| 値 | 意味 |
|---|---|
| ios::app | ファイルへの出力がファイル末尾に付加される |
| ios::ate | ファイルを開くときにファイル末尾までシークする |
| ios::binary | ファイルをバイナリモードでオープンする |
| ios::in | ファイルが入力可能である |
| ios::out | ファイルが出力可能である |
| ios::trunc | 同名の既存ファイルの内容を破棄し，ファイルの長さをゼロにする |

2つ以上の値をORで組み合わせて指定することができます．これらの値の意味を見ておきましょう．

**ios::app**を指定すると，すべてのファイルへの出力がファイル末尾に付加されます．この値は，出力可能なファイルでしか使用できません．**ios::ate**を指定すると，ファイルを開くときファイル末尾までのシークが起こります．このように，ios::ateはend-of-file（ファイルの終わり）をシークしますが，入出力操作自体はファイル内のどこででも行えます．

**ios::in**はファイルが入力可能であること，**ios::out**はファイルが出力可能であることを意味します．

**ios::binary**を指定すると，ファイルがバイナリモードで開かれます．すべてのファイルは，デフォルトではテキストモードで開かれます．テキストモードでは，さまざまな文字変換が起こることがあります．たとえば，キャリッジリターン/ラインフィード文字列は改行に変換されます．しかし，ファイルをバイナリモードで開くときは，そのような文字変換が起こりません．書式設定されたテキストであろうと生データであろうと，あらゆるファイルは，バイナリモードとテキストモードのどちらででも開くことができます．2つの開き方の違いは，唯一，文字変換が起こるかどうかです．

**ios::trunc**を指定すると，同名の既存ファイルの内容が破棄され，ファイルの長さがゼロになります．ofstreamを使用して出力ストリームを作成するとき，すでに同名のファイルがあると，自動的に同じことが起こります．

次のプログラムコードは，testという出力ファイルを開いています．

```
ofstream mystream;
mystream.open("test");
```

open()に与えるmode仮引数は，開かれるストリームの種類に応じて適切なデフォルト値を取るので，上の例で特に値を指定する必要はありません．

open()が失敗すると，ストリームは，ブール式の中では偽と評価されます．これを利用すれば，開く操作が成功したかどうかを次のような文で確認できます．

```
if(!mystream) {
  cout << "Cannot open file.¥n";
  // エラー処理
}
```

一般に，ファイルにアクセスするときは，その前に必ずopen()呼び出しの結果を検査してください．

また，ファイルを開く操作が成功したかどうかの確認には，is_open()関数も使用できます．この関数は，fstream，ifstream，ofstreamのメンバです．is_open()関数のプロトタイプは，次のとおりです．

**is_open()関数**

```
bool is_open( );
```

この関数は，ストリームが開かれたファイルにリンクされていれば真，そうでなければ偽を返します．たとえば，次のプログラムコードは，mystream が現在開いているかどうかを検査します．

```
if(!mystream.is_open()) {
  cout << "File is not open.¥n";
  //  ...
```

open()関数でファイルを開くことは，それ自体はまったく正しい方法ですが，あまり使われません．実際には，ifstream，ofstream，fstreamのどのクラスにも，自動的にファイルを開くコンストラクタ関数があり，それが使用されます．このコンストラクタ関数は，仮引数もデフォルト値もopen()関数と同じです．したがって，最も一般的なファイルを開く方法は，次のようになります．

```
ifstream  mystream("myfile"); // ファイルを入力用に開く
```

先ほど述べたとおり，何らかの理由でファイルが開かれないと，ストリーム変数は条件文の中で偽と評価されます．したがって，コンストラクタ関数でファイルを開く場合も，明示的にopen()を呼ぶ場合も，必ずストリームの値を検査して，ファイルが実際に開いたことを確認してください．

ファイルを閉じるには，close()メンバ関数を使用します．たとえば，mystream というストリームにリンクされているファイルを閉じるには，次の文を使用します．

```
mystream.close();
```

close()関数は仮引数を受け取らず，値を返しません．

入力ファイルの終わりに達したことは，iosのメンバ関数eof()で検出できます．eof()関数のプロトタイプは，次のとおりです．

**eof()関数**

```
bool eof( );
```

この関数は，ファイルの終わりに達していれば真を返し，それ以外は偽を返します．

ファイルが開けば，そこからテキストデータを読み取ったり，書式設定されたテキストデータを書き込んだりすることは，ごく簡単です．コンソール入出力を実行したときと同じ要領で，<<演算子と>>演算子を使用するだけです．ただ，cinとcoutを使用する代わりに，ファイルにリンクされているストリームを使用します．>>と<<を使ってファイルから読み込んだり，ファイルに書き出したりすることは，多少，C の fprintf()関数と fscanf()関数を使うことに似ています．すべての情報は，画面に表示されるとおりの書式でファイルに格納されます．したがって，<< で生成されたファイルは，書式設定されたテキストファイルになりますし，>> で読むファイルは，どれも書式設定されたテキストファイルでなければなりません．このように，>>演算子と<<演算子で操作するファイルは，書式設定された

テキストを含むファイルなので，バイナリモードでなくテキストモードで開くのが一般的です．バイナリモードは，書式不定のファイルに適しています．その種のファイルについては，次の節で説明します．

## 例 9.2 ファイル入出力の基本

1. 次に示すプログラムは，出力ファイルを作成し，そこへ情報を書き込み，ファイルを閉じ，それを入力ファイルとして再度開き，そこから情報を読み取ります．

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream fout("test");  // 出力ファイルを作成する

  if(!fout) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  fout << "Hello!¥n";
  fout << 100 << ' ' << hex << 100 << endl;

  fout.close();
  ifstream fin("test"); // 入力ファイルを開く

  if(!fin) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char str[80];
  int i;

  fin >> str >> i;
  cout << str << ' ' << i << endl;

  fin.close();

  return 0;
}
```

このプログラムの実行後，test の内容を調べると，次のようになっているはずです．

```
Hello!
100 64
```

すでに述べたとおり，ファイル入出力に<<演算子と>>演算子を使用するときは，情報が画面に表示されるとおりの書式になっています．

**2.** ディスク入出力の例をもう1つ見ておきましょう．このプログラムは，キーボードから入力された文字列を読み取り，それをディスクに書き出します．$で始まる文字列が入力されると，プログラムが停止します．このプログラムを使用するには，コマンド行から出力ファイルの名前を指定してください．

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: WRITE <filename>¥n";
    return 1;
  }
  ofstream out(argv[1]); // 出力ファイル

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  char str[80];
  cout << "Write strings to disk, '$' to stop¥n";

  do {
    cout << ": ";
    cin >> str;
    out << str << endl;
  } while (*str != '$');

  out.close();
  return 0;
}
```

**3.** 次のプログラムは，テキストファイルをコピーしながら，すべてのスペースを｜記号に変換します．入力ファイルの終わりの検出にeof()が使われています．また，入力ストリームfinのskipwsフラグがオフになっていることに注意してください．これをオフにしておくと，入力ストリームの先頭にあるスペースのスキップが起こりません．

```
// スペースを|に変換する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CONVERT <input> <output>¥n";
    return 1;
  }

  ifstream fin(argv[1]);    // 入力ファイルを開く
  ofstream fout(argv[2]);   // 出力ファイルを作成する

  if(!fout) {
    cout << "Cannot open output file.¥n";
    return 1;
  }
  if(!fin) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char ch;

  fin.unsetf(ios::skipws);  // スペースをスキップしない
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '|';
    if(!fin.eof()) fout << ch;
  }

  fin.close();
  fout.close();

  return 0;
}
```

**4.** C++の古い入出力ライブラリと新しい標準C++ライブラリの間には，いくつかの違いがあるので，古いプログラムコードを変換する際には注意が必要です．まず，古い入出力ライブラリでは，open()に3つ目の仮引数があって，ファイルの保護モードを指定できました．デフォルトは，通常ファイルです．新しい入出力ライブラリでは，この仮引数がサポートされません．

次に，古いライブラリを使い，fstreamで入出力用のストリームを開くときは，ios::inとios::outという2つのmode値を，ともに明示的に指定しなければなりません．mode

にはデフォルト値がありません．これは，fstream コンストラクタを使うときも，それの open() 関数を使うときも同じです．たとえば，古い入出力ライブラリを使い，open() を呼び出してファイルを入出力用に開くには，次のように呼び出さなければなりません．

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

新しい入出力ライブラリでは，mode 仮引数を指定しなくても，fstream 型のオブジェクトは自動的にファイルを入出力用に開きます．

さらに，古い入出力システムでは，mode 仮引数に ios::nocreate または ios::noreplace を指定できました．ios::nocreate では，既存のファイルがないと open() 関数が失敗します．ios::noreplace では，既存のファイルがあると open() 関数が失敗します．標準 C++ では，これらの値がサポートされていません．

### 練習問題　9.2　ファイル入出力の基本

**1.** テキストファイルをコピーするプログラムを作成しなさい．コピーした文字の数を数え，その結果を表示しなさい．表示される文字数とディレクトリを受け取ったとき，その出力ファイルのサイズとして示される文字数とが異なる理由を説明しなさい．

**2.** 次の情報テーブルを phone というファイルに書き出すプログラムを作成しなさい．

```
Isaac Newton, 415 555-3423
Robert Goddard, 213 555-2312
Enrico Fermi, 202 555-1111
```

**3.** ファイル内の語数を数えるプログラムを作成しなさい．あまり複雑にならないよう，ホワイトスペースで囲まれているものを 1 語と見なします．

**4.** is_open() の働きは何か説明しなさい．

## 9.3　書式不定のバイナリ入出力

先ほどの例で作成したような書式設定されたテキストファイルも，さまざまな状況で便利に使用できることは事実ですが，書式不定のバイナリファイルの持つ柔軟性には欠けます．書式不定のファイルに

は，データのバイナリ表現が含まれています．これは，<<演算子と>>演算子によってデータが変換されてできるような，人間にとって読めるテキストではなく，プログラムが内部的に使用する表現形式です．その意味で，書式不定の入出力は「生の」入出力とも呼ばれます．C++は，さまざまな種類の書式不定ファイル入出力関数をサポートしています．書式不定関数を使用すると，ファイルの読み書きを細かく制御できます．

書式不定入出力関数のうち，最も低水準の関数はget()とput()です．get()では1バイト読み取ることができ，put()では1バイト書き出すことができます．get()はすべての入力ストリームクラスのメンバ関数で，put()はすべての出力ストリームクラスのメンバ関数になっています．get()関数には多くの形式がありますが，最もよく使われる形式は次のとおりです．これに対応するput()形式も示しておきます．

**get()関数とput()関数**

```
istream &get(char &ch);
ostream &put(char ch);
```

get()関数は，対応ストリームから1文字を読み取り，その値をchに読み込んで，ストリームへの参照を返します．ファイルの終わりで読み取りを行おうとすると，戻り時に，呼び出しを行ったストリームが偽と評価されます(式の中で使用されるとき)．put()関数は，chをストリームに書き込み，ストリームへの参照を返します．

データブロックの読み書きには，read()関数とwrite()関数を使用します．やはり，read()関数は入力ストリームクラスのメンバで，write()関数は出力ストリームクラスのメンバです．read()関数とwrite()関数のプロトタイプは，次のとおりです．

**read()関数とwrite()関数**

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

read()関数は，呼び出しを行ったストリームから*num*個のバイトを読み取り，それを*buf*で指し示されるバッファに読み込みます．write()関数は，*buf*で指し示されているバッファから対応ストリームへ*num*個のバイトを書き込みます．streamsize型はある整数型です．streamsize型のオブジェクトは，1回の入出力操作で転送される最大バイト数を保持できます．

*num*個の文字を読み終えないうちにファイルが終わりになると，read()の実行はそこで終わり，バッファには読み取られただけの文字が入ります．この文字数は，メンバ関数gcount()で調べることができます．gcount()関数のプロトタイプは次のとおりです．

**gcount()関数**

```
streamsize gcount( );
```

この関数は，最後の書式不定入力操作で読み取られた文字数を返します．

書式不定ファイル関数を使用するときは，ファイルをバイナリ操作用に(テキスト操作用でなく)開くのが一般的です．その理由は明らかで，ios::binary を指定すれば，文字変換を起こさずに済むからです．これは，整数，浮動小数点数，ポインタなど，データのバイナリ表現がファイルに格納されているときには重要なことです．しかし，ファイルにテキストしか含まれていないことが確実であれば，そのファイルをテキストモードで開いてから，書式不定関数で操作することには，何の不都合もありません．しかし，一部で文字変換が起こるかもしれないことに注意してください．

## 例 9.3　書式不定のバイナリ入出力

**1.** 次のプログラムは，あらゆるファイルの内容を画面上に表示します．get()関数を使います．

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: PR <filename>¥n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.¥n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();
```

```
  return 0;
}
```

2. 次のプログラムは，put()を使用して文字をファイルに書き出します．ユーザーが$記号を入力するまで，書き出しを続けます．

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: WRITE <filename>¥n";
    return 1;
  }

  ofstream out(argv[1], ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open file.¥n";
    return 1;
  }

  cout << "Enter a $ to stop¥n";
  do {
    cout << ": ";
    cin.get(ch);
    out.put(ch);
  } while (ch!='$');

  out.close();

  return 0;
}
```

このプログラムは，cinから文字を読み取るのにget()を使用しています．このため，始めにスペースがあっても，破棄されることはありません．

3. 次のプログラムはwrite()を使い，double型の値と文字列をtestというファイルに書き出します．

```
#include <iostream>
#include <fstream>
#include <cstring>
```

```
using namespace std;

int main()
{
  ofstream out("test", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  double num = 100.45;
  char str[] = "This is a test";

  out.write((char *) &num, sizeof(double));
  out.write(str, strlen(str));

  out.close();

  return 0;
}
```

メモ

write()呼び出しの内部で(char *)への型キャストを行っています．文字配列として定義されていないバッファを出力するときには，これが必要です．C++ では型検査が強力であるため，ある型のポインタから別の型のポインタへの変換が自動的には行われません．

**4.** 次のプログラムはread()を使い，例3のプログラムで作成したファイルを読み取ります．

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("test", ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  double num;
  char str[80];

  in.read((char *) &num, sizeof(double));
```

```
  in.read(str, 14);
  str[14] = '¥0'; // ヌル終端文字列
  cout << num << ' ' << str;

  in.close();

  return 0;
}
```

上の例のプログラムと同様，read()内での型キャストが必要です．C++は，ある型のポインタを別の型のポインタに自動的には変換しません．

**5.** 次のプログラムでは，最初にdouble値の配列をファイルに書き出してから，それを読み戻します．読み込んだ文字の数も報告します．

```
// gcount()の働きを示す
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out("test", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  double nums[4] = {1.1, 2.2, 3.3, 4.4 };
  out.write((char *) nums, sizeof(nums));
  out.close();

  ifstream in("test", ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  in.read((char *) &nums, sizeof(nums));

  int i;
  for(i=0; i<4; i++)
    cout << nums[i] << ' ';

  cout << '¥n';
```

```
  cout << in.gcount() << " characters read¥n";
  in.close();

  return 0;
}
```

## 練習問題 9.3 書式不定のバイナリ入出力

1. 9.2 節の練習問題 1 と 3 の答えを，get()，put()，read()，write() を使って書き直しなさい(適切と思うもののどれを使ってもかまいません).

2. 次のクラスの内容をファイルに出力するプログラムを作成しなさい．必要な挿入関数を作成しなさい.

```
class account {
  int custnum;
  char name[80];
  double balance;
public:
  account(int c, char *n, double b)
  {
    custnum = c;
    strcpy(name, n);
    balance = b;
  }
  // ここに挿入子を作成する
};
```

# 9.4　その他の書式不定入出力関数

get() 関数には，前節で見た形式のほかに，いくとおりものオーバーロード形式があります．そのうち最もよく使われる形式のプロトタイプを，3 とおり示しておきます.

get() 関数

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get( );
```

最初の形式は，*buf* が指す配列に文字を読み込みます．*num*-1 個の文字が読まれるか，改行が見つかるか，ファイルの終わりに達すると，読み込みが終わります．*buf* が指す配列は，get() の働きによ

りヌルで終端されます．入力ストリーム中に改行文字が見つかっても，これは抽出されません．次の入力操作までストリームに残ります．

2つ目の形式は，*buf*が指している配列に文字を読み込みます．*num*-1個の文字が読まれるか，*delim*で指定された文字が見つかるか，ファイルの終わりに達すると，読み込みが終わります．*buf*が指す配列は，get()の働きによりヌルで終端されます．入力ストリーム中に区切り文字が見つかっても，これは抽出されません．次の入力操作までストリームに残ります．

get()の3つ目のオーバーロード形式は，ストリーム中の次の1文字を返します．ファイルの終わりに達すると，EOFを返します．この形式のget()は，Cのgetc()関数に似ています．

入力を実行する関数には，ほかにgetline()があります．これは，各入力ストリームクラスのメンバです．getline()関数のプロトタイプは，次のとおりです．

getline()関数

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

最初の形式は，*buf*が指す配列に文字を読み込みます．*num*-1個の文字が読まれるか，改行が見つかるか，ファイルの終わりに達すると，読み込みが終わります．*buf*が指す配列は，getline()の働きによりヌルで終端されます．入力ストリーム中に改行文字が見つかると，抽出されますが，*buf*には読み込まれません．

2つ目の形式は，*buf*が指す配列に文字を読み込みます．*num*-1個の文字が読まれるか，*delim*で指定された文字が見つかるか，ファイルの終わりに達すると，読み込みが終わります．*buf*が指す配列は，getline()の働きによりヌルで終端されます．入力ストリーム中に区切り文字が見つかると，抽出されますが，*buf*には読み込まれません．

ご覧のとおり，getline()の2形式は，get()の2形式，get(*buf*,*num*)とget(*buf*,*num*,*delim*)とほとんど同じです．どちらも入力から文字を読み取り，*buf*で指し示される配列にその文字を読み込みます．*num*-1個の文字が読まれるか，区切り文字が見つかるか，ファイルの終わりに達すると，読み込みが終わります．get()とgetline()の違いは，getline()が入力ストリームから区切り文字を読み，それを取り除くのに対し，get()はそれを行わないことです．

peek()を使用すれば，入力ストリームから次の1文字を読むだけで，ストリームから取り除かずにおくことができます．peek()入力ストリームクラスのメンバです．peek()関数のプロトタイプは次のとおりです．

peek()関数

```
int peek( );
```

この関数は入力ストリーム中の次の１文字を返し，ファイルの終わりに達するとEOFを返します．

putback()を使用すれば，ストリームから読んだ最後の1文字を，またストリームに戻すことができます．putback()は入力ストリームクラスのメンバで，putback()関数のプロトタイプは次のとおりです．

putback()関数

```
istream &putback(char c);
```

*c*は，読み込まれた最後の１文字です．

出力を実行しても，ストリームにリンクされた物理デバイスにデータが直ちに書き出されるわけではありません．情報はいったん内部バッファに格納され，バッファが満杯になると，ディスクに書き出されます．しかしflush()を使えば，バッファが満杯になる前に，強制的に情報をディスクに書き出すことができます．flush()は出力ストリームクラスのメンバで，flush()関数のプロトタイプは次のとおりです．

flush()関数

```
ostream &flush( );
```

プログラムを悪環境(たとえば，頻繁に停電が起こるなど)で使用するときは，flush()の使用を考えるとよいでしょう．

## 例 9.4　その他の書式不定入出力関数

**1.**　ご存じのとおり，>>を使って文字列を読み取るときは，ホワイトスペース文字が現れると，読み取りが終了します．このため，スペースを含んでいる文字列の読み取りには使用できません．しかし，次のプログラムで示すように，getline()を使えばこの問題を解決できます．

```
// getline()を使用して，スペースを含む文字列を読み取る
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char str[80];
```

```
  cout << "Enter your name: ";
  cin.getline(str, 79);

  cout << str << '¥n';

  return 0;
}
```

このプログラムでは，getline()が区切り文字として改行を使用しています．このため，このgetline()は，標準のgets()関数に似た振る舞いをします．

**2.** 現実のプログラミング状況では，どのような種類の情報が入力されるのかがいつもわかっているわけではありません．そのような状況にも，peek()関数とputback()関数で容易に対処できます．次のプログラムでその感覚がわかります．これは，ファイルから文字列または整数を読み取ります．文字列と整数はどのような順序で発生するかわかりません．

```
// peek()の使用例
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main()
{
  char ch;

  ofstream out("test", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }
  char str[80], *p;

  out << 123 << "this is a test" << 23;
  out << "Hello there!" << 99 << "sdf" << endl;
  out.close();

  ifstream in("test", ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }
```

```
  do {
    p = str;
    ch = in.peek(); // 次の文字の種類を見る

    if(isdigit(ch)) {
      while(isdigit(*p=in.get())) p++; // 整数を読み取る
      in.putback(*p); // 文字をストリームに返す
      *p = '¥0'; // 文字列をヌルで終了させる
      cout << "Integer: " << atoi(str);
    }
    else if(isalpha(ch)) { // 文字列を読み取る
      while(isalpha(*p=in.get())) p++;
      in.putback(*p); // 文字をストリームに返す
      *p = '¥0'; // 文字列をヌルで終了させる
      cout << "String: " << str;
    }
    else in.get(); // 無視
    cout << '¥n';
  } while(!in.eof());

  in.close();
  return 0;
}
```

### 練習問題 9.4 その他の書式不定入出力関数

1. 例1のプログラムを，getline()の代わりにget()を使って書き直しなさい．プログラムの働きは変わるかどうか調べなさい．
2. テキストファイルを一度に1行ずつ読み取り，どの行も画面に表示するプログラムを作成しなさい．getline()を使用してください．
3. どのような場合にflush()を呼ぶことが適切なのか考えなさい．

## 9.5 ランダムアクセス

C++の入出力システムでは，seekg()関数とseekp()関数によってランダムアクセスを実行します．seekg()は入力ストリームクラスのメンバ，seekp()は出力ストリームクラスのメンバです．seekg()関数とseekp()関数では次の形式が最もよく使われます．

seekg()関数と seekp()関数

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

off_type は ios で定義されている整数型で，*offset* が取り得る最大有効値を含むことができます．seekdir は ios で定義されている列挙型で，次の値を取ります．

**表9-2　seekdir が取る値**

| 値 | 意味 |
|---|---|
| ios::beg | 先頭からシークする |
| ios::cur | 現在位置からシークする |
| ios::end | 終わりからシークする |

C++では，ファイルに2つのポインタが関連付けられていて，これを入出力システムが管理します．1つは**getポインタ**で，ファイルのどこで次の入力操作が行われるかを指定します．もう1つは**putポインタ**で，ファイルのどこで次の出力操作が行われるかを指定します．入力操作または出力操作が行われると，そのたびに当該ポインタが自動的かつ順次に前進します．しかし，seekg()関数とseekp()関数を使用すれば，ファイルにランダムアクセスを行うことができます．

seekg()関数は，対応ファイルの現getポインタを，指定されたoriginからoffsetバイト数だけ移動させます．seekp()関数は，対応ファイルの現putポインタを，指定されたoriginからoffsetバイト数だけ移動させます．

一般に，seekg()とseekp()でアクセスされるファイルは，バイナリファイル操作用に開いておいてください．こうすると文字変換が起こらず，ファイル中の項目の見かけの位置が変わることもありません．

各ファイルポインタの現在位置は，次のメンバ関数で調べられます．

tellg()関数と tellp()関数

```
pos_type tellg( );
pos_type tellp( );
```

pos_type は，ios で定義されている整数型で，ファイル位置を指し示す最大値を保持できます．

seekg()とseekp()にはオーバーロード形式があり，それぞれtellg()とtellp()の戻り値で指定された位置へファイルポインタを移動させます．seekg()関数とseekp()関数のプロトタイプは次のとおりです．

**seekg()関数と seekp()関数**

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

## 例 9.5 ランダムアクセス

**1.** 次に示すのは，seekp()関数の働きを示すプログラムです．このプログラムでは，ファイル中の特定の1文字を変更します．コマンド行からファイル名を指定し，続いてファイル中の何バイト目の文字を変更したいか，それをどの文字に変更したいのかを指定します．ファイルが読み書き操作用に開かれていることに注意してください．

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Usage: CHANGE <filename> <byte> <char>¥n";
    return 1;
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open file.¥n";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);
  out.put(*argv[3]);
  out.close();

  return 0;
}
```

**2.** 次のプログラムはseekg()を使い，ファイルの中ほどにgetポインタを位置付け，その位置以後のファイル内容を表示します．ファイル名と読み取り開始位置は，コマンド行から指定します．

```
// seekg()の働きを示す
#include <iostream>
```

```
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: LOCATE <filename> <loc>¥n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();
  return 0;
}
```

## 練習問題　9.5　ランダムアクセス

**1.** テキストファイルを逆方向に表示していくプログラムを作成しなさい．

ヒント　よく考えてから，プログラムの作成に取りかかってください．想像するほど難しくありません．

**2.** テキストファイル内の2文字を1組として順序を入れ替えるプログラムを作成しなさい．たとえば，ファイルに「1234」が含まれているとき，プログラムの実行後にそれが「2143」になるようにします(あまり複雑にならないよう，ファイルには偶数個の文字が含まれているとします)．

# 9.6 入出力状態のチェック

C++の入出力システムは，各入出力操作の結果についての状態情報を保持しています．入出力ストリームの現在の状態は，iostate型オブジェクトに記述されています．iostateはiosで定義されている列挙型で，次のメンバを含んでいます．

**表9-3 iostate型オブジェクトのメンバ**

| 名前 | 意味 |
|---|---|
| goodbit | エラーは起こらなかった |
| eofbit | ファイルの終わりに達した |
| failbit | 致命的ではない入出力エラーが起こった |
| badbit | 致命的な入出力エラーが起こった |

古いコンパイラでは，入出力状態フラグがiostate型オブジェクトではなくintに保持されます．

入出力状態情報を取得する方法は2とおりあります．1つは，iosのメンバであるrdstate()関数を呼ぶ方法です．プロトタイプは次のとおりです．

rdstate()関数を呼ぶ

```
iostate rdstate( );
```

この関数はエラーフラグの現状態を返します．上のフラグリストから想像できるとおり，エラーがなければ，rdstate()はgoodbitを返します．それ以外のときはエラーフラグを返します．

エラーの有無を調べるもう1つの方法は，次のiosメンバ関数を1つ以上使用する方法です．

iosメンバ関数

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

eof()関数については9.2節で説明しました．bad()関数は，badbitがオンのとき真を返します．fail()関数は，failbitがオンのとき真を返します．good()関数は，エラーがないとき真を返します．それ以外は，どの関数も偽を返します．

エラーが起こると，それをクリアしない限り，プログラムを続行できないことがあります．エラーをクリアするには，iosのclear()メンバ関数を使用します．プロトタイプは次のとおりです．

clear()メンバ関数

```
void clear(iostate flags = ios::goodbit);
```

*flags* をgoodbit（デフォルト値）にすると，すべてのエラーフラグがクリアされます．それ以外のときは，flags を希望の値に設定してください．

## 例 9.6 入出力状態のチェック

**1.** 次に示すのは，rdstate()の使い方を示すプログラムです．テキストファイルの内容を表示します．エラーが起こると，checkstatus()を用いてそのエラーを報告します．

```
#include <iostream>
#include <fstream>
using namespace std;

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: DISPLAY <filename>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char c;
  while(in.get(c)) {
    cout << c;
    checkstatus(in);
  }

  checkstatus(in);  // 最終状態をチェックする
  in.close();

  return 0;
}

void checkstatus(ifstream &in)
```

```
{
  ios::iostate i;
  i = in.rdstate();

  if(i & ios::eofbit)
    cout << "EOF encountered¥n";
  else if(i & ios::failbit)
    cout << "Non-Fatal I/O error¥n";
  else if(i & ios::badbit)
    cout << "Fatal I/O error¥n";
}
```

このプログラムは，少なくとも1つの「エラー」を必ず報告します．whileループが終わって，最後のcheckstatus()呼び出しが行われると，当然，EOFになったことが報告されます．

**2.** 次のプログラムはテキストファイルを表示します．good()を使用して，ファイルエラーを検出します．

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "PR: <filename>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    // エラーの有無をチェック
    if(!in.good() && !in.eof()) {
      cout << "I/O Error...terminating¥n";
      return 1;
    }
    cout << ch;
```

```
    }

    in.close();

    return 0;
}
```

**練習問題　9.6　入出力状態のチェック**

**1.**　前節の練習問題の解答にエラー検査を付け加えなさい．

# 9.7　カスタマイズされた入出力とファイル

前章で，独自クラスに関して挿入演算子と抽出演算子をオーバーロードする方法を学びました．そこではコンソール入出力しか実行しませんでしたが，すべてのC++ストリームは同じなので，たとえば，画面への出力に用いられるオーバーロード挿入関数は，そのまま(何の変更も必要とせず)ファイルへの出力にも使用できます．これは，入出力に対するC++ の取り組み方を特徴付ける，最も重要かつ便利な機能の1つです．

前章で述べたとおり，オーバーロードされた挿入子と抽出子，さらには入出力マニピュレータは，汎用的に書かれている限り，どのようなストリームにでも使用できます．特定ストリームを入出力関数に「ハードコード」してしまうと，その関数はそのストリームにしか使えなくなります．このため，入出力関数はできる限り汎用的に作成するようお勧めします．

## 例　9.7　カスタマイズされた入出力とファイル

**1.**　次のプログラムで，coord クラスは<< 演算子と >> 演算子をオーバーロードしています．この演算子関数は，画面への書き出しにもファイルへの書き込みにも使用できることに注意してください．

```
#include <iostream>
#include <fstream>
using namespace std;

class coord {
  int x, y;
public:
  coord(int i, int j) { x = i; y = j; }
```

```
  friend ostream &operator<<(ostream &stream, coord ob);
  friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ' ' << ob.y << '¥n';
  return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
  stream >> ob.x >> ob.y;
  return stream;
}

int main()
{
  coord o1(1, 2), o2(3, 4);

  ofstream out("test");

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  out << o1 << o2;
  out.close();

  ifstream in("test");

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  coord o3(0, 0), o4(0, 0);
  in >> o3 >> o4;
  cout << o3 << o4;
  in.close();

  return 0;
}
```

**2.** すべての入出力マニピュレータは，ファイルにも使用できます．たとえば，次に示すのは，この章ですでに取り上げたプログラムを書き直したものです．画面に書き出すマニピュレータを，そのままファイルへの書き出しにも使用しています．

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

// 警告
ostream &atn(ostream &stream)
{
  stream << "Attention: ";
  return stream;
}

// 注意書き
ostream &note(ostream &stream)
{
  stream << "Please Note: ";
  return stream;
}

int main()
{
  ofstream out("test");

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  // 画面への書き出し
  cout << atn << "High voltage circuit¥n";
  cout << note << "Turn off all lights¥n";

  // ファイルへの書き出し
  out << atn << "High voltage circuit¥n";
  out << note << "Turn off all lights¥n";
  out.close();

  return 0;
}
```

## 練習問題　9.7　カスタマイズされた入出力とファイル

**1.** 前章のプログラムをディスクファイル用に書き直し，いろいろと試してみなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるどうか確認しましょう．

**1.** 3個のタブを出力し，フィールド幅を20桁に設定する出力マニピュレータを作成しなさい．そのマニピュレータが実際に機能するかどうか試してみなさい．

**2.** アルファベット以外の文字を読んだときは，それを破棄する入力マニピュレータを作成しなさい．最初のアルファベットを読んだら，それを入力ストリームに返してから，戻ります．このマニピュレータをfindalphaという名前にします．

**3.** テキストファイルをコピーするプログラムを作成しなさい．コピーしながら，すべての文字の大文字と小文字を逆に変換します．

**4.** テキストファイルを読み取りながら，アルファベットの各文字の出現回数を数えるプログラムを作成しなさい．

**5.** 練習問題3と4の解答に，完全なエラー検査機能を付け加えなさい(すでに付け加えてあるならこの問題は無視してください)．

**6.** getポインタを位置付ける関数は何か，putポインタを位置付ける関数は何か，それぞれ説明しなさい．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

**1.** 次に示すのは，前章のinventoryクラスを書き直したものです．store()関数とretrieve()関数を完成させるプログラムを作成しなさい．次に，数品目の小さな在庫管理ファイルをディスク上に作成し，ランダム入出力を用いて，レコード番号を指定することで当該品目についての情報を表示できるようにしなさい．

```
#include <fstream>
#include <iostream>
#include <cstring>
```

```
using namespace std;

#define SIZE 40

class inventory {
  char item[SIZE]; // 品名
  int onhand; // 在庫数量
  double cost;  // 原価
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  void store(fstream &stream);
  void retrieve(fstream &stream);
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " on hand at $" << ob.cost << '¥n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
  cout << "Enter item name: ";
  stream >> ob.item;
  cout << "Enter number on hand: ";
  stream >> ob.onhand;
  cout << "Enter cost: ";
  stream >> ob.cost;

  return stream;
}
```

**2.** （特別にやる気のある人向け）
文字のstackクラスを作成しなさい．これは，文字をメモリ中の配列でなくディスクファイルに格納します．

# 10

# 仮想関数

この章の内容

この章では，C++が持つ重要な機能の1つである仮想関数を見ていきます．仮想関数が重要なのは，実行時ポリモーフィズムのサポートに使われるからです．C++では，ポリモーフィズムが2とおりの方法でサポートされます．1つはコンパイル時のサポートで，オーバーロードされた演算子や関数によって行います．もう1つは実行時のサポートで，仮想関数を使用します．柔軟性の高さでは実行時ポリモーフィズムの方が優れています．

仮想関数と実行時ポリモーフィズムの基盤には，派生クラスへのポインタがあります．この章では，このポインタの説明から始めます．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. 数値を科学的記数法で表示するためのマニピュレータを作成しなさい．Eには大文字を使用します．

2. テキストファイルをコピーするプログラムを作成しなさい．コピーしながら，すべてのタブを正しい数のスペースに置き換えます．

3. コマンド行から指定された単語を，テキストファイル内から探し出すプログラムを作成しなさい．その単語が何個見つかったかを表示します．あまり複雑にならないよう，ホワイトスペースで囲まれたものはすべて単語と見なします．

4. putポインタを，outというストリームにリンクしているファイルの234バイト目に位置付ける文を作成しなさい．

5. C++の入出力システムでステータス情報を報告する関数は何かを説明しなさい．

6. Cの入出力関数ではなくC++の入出力関数を使うことの利点を1つ挙げなさい．

# 10.1　派生クラスへのポインタ

C++ポインタについては第4章で説明しましたが，仮想関数に関係する特殊な側面についての説明は避けてきました．それは，「基本クラスへのポインタとして宣言されたポインタは，その基本クラスか

ら派生したどのクラスを指し示すのにも使用できる」ということです．たとえば，base と derived という 2 つのクラスがあって，derived は base を継承しているとします．その場合，次に示す文はいずれも正しいです．

```
base *p; // baseクラスのポインタ

base base_ob; // base型オブジェクト
derived derived_ob; // derived型オブジェクト

// pは，もちろん，baseオブジェクトを指すことができる
p = &base_ob; // pは，baseオブジェクトを指し示す

// pがderivedオブジェクトを指しても，エラーにはならない
p = &derived_ob; // pは，derivedオブジェクトを指し示す
```

コメントに示しているとおり，基本クラスのポインタは，その基本クラスから派生したどのクラスのオブジェクトをも指すことができます．型の不一致エラーは生じません．

基本クラスのポインタで派生オブジェクトを指すことはできますが，アクセスできるのは，派生オブジェクトのメンバのうち基本クラスから継承されたメンバに限られます．これは，基本クラスのポインタには，基本クラスについての知識しかないためです．派生クラスによって追加されたメンバについては何も知りません．

基本クラスのポインタが派生オブジェクトを指すことは認められますが，その逆は認められません．派生クラスのポインタで，基本クラスのオブジェクトにアクセスすることはできません(型キャストを使えば，この制約を克服できますが，お勧めできません)．

最後にもう1つ．ポインタがどのデータ型を指すものとして宣言されているかにより，ポインタ算術が異なることに注意してください．たとえば，基本クラスのポインタで派生オブジェクトを指し示しているとき，そのポインタの値に1を加えても，次の派生オブジェクトを指すというわけにはいきません．実際には，次の(と，ポインタが見なす)基本オブジェクトを指し示します．この点に十分注意してください．

## 例 10.1　派生クラスへのポインタ

**1.** 次に，基本クラスのポインタを使って派生クラスにアクセスする短いプログラムを示します．

```
// 派生クラスを指すポインタの実例
#include <iostream>
using namespace std;
```

```
class base {
  int x;
public:
  void setx(int i) { x = i; }
  int getx() { return x; }
};

class derived : public base {
  int y;
public:
  void sety(int i) { y = i; }
  int gety() { return y; }
};

int main()
{
  base *p;       // base型へのポインタ
  base b_ob;     // baseクラスのオブジェクト
  derived d_ob; // derivedクラスのオブジェクト

  // pを使用して，baseクラスのオブジェクトにアクセスする
  p = &b_ob;
  p->setx(10);  // baseクラスのオブジェクトにアクセス
  cout << "Base object x: " << p->getx() << '¥n';

  // pを使用して，derivedクラスのオブジェクトにアクセスする
  p = &d_ob;     // derivedクラスのオブジェクトを指す
  p->setx(99);  // derivedクラスのオブジェクトにアクセス

  // pによるyの設定はできないので，直接行う
  d_ob.sety(88);
  cout << "Derived object x: " << p->getx() << '¥n';
  cout << "Derived object y: " << d_ob.gety() << '¥n';

  return 0;
}
```

この例には，確かに派生クラスを指すポインタが使われてはいますが，この例に見る方法で基本クラスのポインタを使用しても実際的な意味はありません．しかし，次の節を読めば，基本クラスポインタで派生オブジェクトを指すことが，なぜそれほど重要なのかがわかります．

**練習問題 10.1 派生クラスへのポインタ**

1. 上の例を使っていろいろ試してみなさい．たとえば，派生ポインタを宣言し，それを使って基本クラスのオブジェクトにアクセスしたらどうなるか説明しなさい．

# 10.2　仮想関数の概要

**仮想関数**(virtual function)とは，基本クラス内で宣言され，派生クラス内で再定義されるメンバ関数です．仮想関数を作成するには，関数宣言の前にvirtualというキーワードを置きます．仮想関数を含むクラスが継承されると，継承した派生クラスはその仮想関数を自分自身に関して再定義します．要するに，仮想関数とは，ポリモーフィズムの根底にある「1つのインターフェイス，複数のメソッド」という原則を具象化したものです．基本クラス内の仮想関数は，その関数へのインターフェイスの形を定義します．個々の派生クラスによる仮想関数の再定義により，同派生クラスにおけるその関数の動作が実装されます．つまり，再定義とは具体的なメソッドを作成することと言えます．派生クラスで仮想関数を再定義するときは，キーワードvirtualが不要です．

仮想関数の呼び出し方は，ほかのメンバ関数と変わりません．しかし，仮想関数の特長は，ポインタを通じて仮想関数を呼び出したときの動作にあります．また，これが実行時ポリモーフィズムのサポートも実現しています．前の節で，基本クラスポインタを使って派生クラスのオブジェクトを指すことができることがわかりました．基本クラスのポインタが，仮想関数を含む派生オブジェクトを指し，その仮想関数がそのポインタを通じて呼び出されると，C++は，ポインタで指し示されているオブジェクトの型に基づいて，その関数のどのバージョンを実行するかの判断が実行時に行われます．言い換えると，仮想関数のどのバージョンを実行するかは，呼び出し時に指されているオブジェクトの型によって決まります．したがって，仮想関数を含んでいる基本クラスから複数のクラスが派生しているとき，基本ポインタがさまざまなオブジェクトを指すのに応じて，仮想関数のさまざまなバージョンが実行されます．これが，実行時ポリモーフィズムを実現するしくみです．この意味で，仮想関数を含んでいるクラスを**ポリモーフィッククラス**(polymorphic class)と呼びます．

**例 10.2 仮想関数の概要**

1. 次は，仮想関数を使用する短いプログラムです．

```
// 仮想関数を使用する簡単な例
#include <iostream>
```

```
using namespace std;

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Using base version of func(): ";
    cout << i << '¥n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Using derived1's version of func(): ";
    cout << i*i << '¥n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  void func()
  {
    cout << "Using derived2's version of func(): ";
    cout << i+i << '¥n';
  }
};

int main()
{
  base *p;
  base ob(10);

  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // baseのfunc()を使用する
  p = &d_ob1;
  p->func(); // derived1のfunc()を使用する
  p = &d_ob2;
  p->func(); // derived2のfunc()を使用する
```

```
    return 0;
}
```

このプログラムの出力は次のようになります．

```
Using base version of func(): 10
Using derived1's version of func(): 100
Using derived2's version of func(): 20
```

派生クラスにおける仮想関数の再定義は，一見，関数オーバーロードに似ていると思われるかもしれませんが，この2つはまったくの別物です．まず，オーバーロードされた関数は，元の関数とは仮引数の型や個数が違わなければなりません．これに対して仮想関数の再定義では，仮引数の型と個数，さらには戻り型がまったく同じでなければなりません(仮想関数の再定義で仮引数の個数または型を変更すると，それは単に関数のオーバーロードになり，仮想関数としての性質を失います)．また，仮想関数がクラスメンバでなければならない点も，関数のオーバーロードとは違います．さらに，デストラクタは仮想関数でありえますが，コンストラクタは仮想関数になれません．関数オーバーロードと仮想関数の再定義にはこのような違いがあることから，仮想関数の再定義を**オーバーライド**(overriding）と呼ぶことがあります．

上の例では，ご覧のとおり3つのクラスを作成しています．baseクラスが仮想関数func()を定義しており，このクラスをderived1とderived2が継承しています．この3つのクラスはどれもfunc()をオーバーライドし，独自に実装しています．main()の内部では，base型，derived1型，derived2型のオブジェクトとともに基本クラスポインタpが宣言されています．まず，pにob（base型のオブジェクト)のアドレスを代入します．このpを使ってfunc()を呼ぶと，baseで定義されたfunc()が使用されます．次に，pにd_ob1のアドレスを代入し，またfunc()を呼びます．どの仮想関数が呼ばれるかを決めるのは，指し示されているオブジェクトの型なので，今度実行されるのはderived1でオーバーライドされたfunc()です．最後に，pにd_ob2のアドレスを代入して，func()を呼び出すと，今度はderived2で定義されたfunc()が実行されます．

この例の要点は2つあります．1つは，基本クラスポインタを通じて仮想関数にアクセスするとき，オーバーライドされた仮想関数のどのバージョンが実行されるかは，指し示されたオブジェクトの型で判断されることです．もう1つは，その判断が実行時に行われることです．

**2.** 仮想関数は，継承順の階層構造になっています．また，仮想関数をオーバーライドしていない派生クラスでは，基本クラスで定義されたままの関数が使用されます．たとえば，次に示すのは，上のプログラムを少し手直ししたものです．

```
// 仮想関数は階層構造を成す
#include <iostream>
using namespace std;

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Using base version of func(): ";
    cout << i << '¥n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Using derived1's version of func(): ";
    cout << i*i << '¥n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  // derived2はfunc()をオーバーライドしていない
};

int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // baseのfunc()を使用する
  p = &d_ob1;
  p->func(); // derived1のfunc()を使用する
  p = &d_ob2;
```

```
    p->func(); // baseのfunc()を使用する

    return 0;
  }
```

このプログラムの出力は次のようになります．

```
Using base version of func(): 10
Using derived1's version of func(): 100
Using base version of func(): 10
```

このプログラムでは，derived2がfunc()をオーバーライドしていません．pにd_ob2のアドレスを代入し，func()を呼び出すと，baseの関数定義が使用されます．これは，クラス階層においてbaseがderived2の1つ上に位置するからです．一般に，派生クラスが仮想関数をオーバーライドしていないときは，基本クラスの定義が使用されます．

**3.** 次の例は，実行時に起こるランダムなイベントに，仮想関数がどう対応するかを示しています．このプログラムでは，標準乱数ジェネレータrand()から返される値に基づいて，d_ob1とd_ob2の一方を選択します．どちらのfunc()が実行されるかは，実行時にならないとわからないことに注意してください(どのfunc()が呼ばれるか，コンパイル時に知ることは不可能です)．

```
/* 実行時に起こるランダムなイベントに，
   仮想関数を使って対応する例 */
#include <iostream>
#include <cstdlib>
using namespace std;

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Using base version of func(): ";
    cout << i << '¥n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
```

```
        cout << "Using derived1's version of func(): ";
        cout << i*i << '¥n';
      }
    };

    class derived2 : public base {
    public:
      derived2(int x) : base(x) {}
      void func()
      {
        cout << "Using derived2's version of func(): ";
        cout << i+i << '¥n';
      }
    };

    int main()
    {
      base *p;
      derived1 d_ob1(10);
      derived2 d_ob2(10);
      int i, j;

      for(i=0; i<10; i++) {
        j = rand();
        if((j%2)) p = &d_ob1; // 奇数のときはd_ob1を使用
        else p = &d_ob2;      // 偶数のときはd_ob2を使用
        p->func();            // 適切な関数を呼び出す
      }

      return 0;
    }
```

**4.** 次の例は，仮想関数の使い方としてはかなり実用的です．まず，areaという基本クラスを作成し，図形の2つの寸法を保持します．また，getarea()という仮想関数も宣言しています．getarea()は，派生クラスでオーバーライドされ，その派生クラスで定義されている図形の面積を返します．このプログラムの場合，基本クラスのgetarea()宣言はインターフェイスの性質を決めるだけのもので，実際の実装は，この基本クラスを継承する各クラスに任されます．この例では，三角形と長方形の面積を計算してみます．

```
// 仮想関数を使用して，インターフェイスを定義する
#include <iostream>
using namespace std;

class area {
  double dim1, dim2; // 図形の寸法
```

```
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea()
  {
    cout << "You must override this function¥n";
    return 0.0;
  }
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;
    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

int main()
{
  area *p;
  rectangle r;
  triangle t;

  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);

  p = &r;
```

```
    cout << "Rectangle has area: " << p->getarea() << '¥n';

    p = &t;
    cout << "Triangle has area: " << p->getarea() << '¥n';

    return 0;
  }
```

areaでのgetarea()定義はただのプレースホルダで，実質的な機能を持たないことに注意してください．areaは具体的な図形とはリンクされていないため，area内のgetarea()には意味のある定義はできません．getarea()を実際に使用しようと思えば，派生クラスでオーバーライドしなければなりません．次の節では，それを強制的にさせる方法を見ます．

**練習問題　10.2　仮想関数の概要**

1. numという基本クラスを作成するプログラムを作成しなさい．このクラスは，整数値を1つ持ち，shownum()という仮想関数を含んでいます．numを継承する2つの派生クラスouthexとoutoctを作成し，それぞれがshownum()をオーバーライドして，整数値を16進数と8進数で表示するようにしなさい．

2. distという基本クラスを作成するプログラムを作成しなさい．distは，2地点間の距離をdouble型の変数に格納します．また，その距離を移動するのにかかる時間を出力する，trav_time()という仮想関数を含んでいます．距離はマイル単位で表されていて，速度は時速60マイルとします．次に，metricという派生クラスを作成します．trav_time()をオーバーライドして，ここでは距離がキロメートル単位，速度が時速100キロメートルとして，移動時間を出力します．

# 10.3　仮想関数の詳細

10.2節の例4で示したとおり，基本クラスで仮想関数を宣言しても，それが特に意味のある操作を実行しないことがあります．このような状況は珍しくありません．基本クラスは，それ単独で完全なクラスを定義するものではなく，単に中心となる1組のメンバ関数と変数を用意して，あとで必要なものを派生クラスに用意させることが多いからです．基本クラスの仮想関数が何も意味のある操作を実行しないということは，派生クラスがこの関数を必ずオーバーライドしなければならないことを意味します．必ずオーバーライドが行われるように，C++では純粋仮想関数をサポートしています．

純粋仮想関数には，基本クラスに関する定義がありません．関数のプロトタイプだけが示されます．純粋仮想関数を作成するには，次の一般形式を使用します．

**純粋仮想関数の一般形式**

```
virtual type func-name( parameter-list ) = 0;
```

この宣言で重要なのは，関数を0に等しいと設定していることです．コンパイラはこれにより，この関数には基本クラスに関する本体が存在しないことを知ります．仮想関数が純粋仮想関数の形で宣言されると，すべての派生クラスはそれを必ずオーバーライドしなければなりません．オーバーライドしないと，コンパイルエラーになります．仮想関数を純粋にすることは，派生クラスが必ず独自に再定義を行うよう保証するための方法です．

少なくとも1個の純粋仮想関数を含んでいるクラスを，**抽象クラス**（abstract class）と呼びます．抽象クラスは，本体を持たない関数を少なくとも1個含んでいるため，技術的には不完全な型であり，そのクラスのオブジェクトを作成することはできません．つまり，抽象クラスは，継承されて初めて存在できるクラスです．自力で存在するように意図されていませんし，実際にできません．しかし，それでも抽象クラスへのポインタは作成できます．基本クラスポインタを使用しなければ，実行時ポリモーフィズムは実現されないため，これは当然のことです（抽象クラスへの参照も認められます）．

仮想関数が継承されると，その仮想性も継承されます．つまり，派生クラスが基本クラスから仮想関数を継承し，さらに別の派生クラスの基本クラスとして使用される場合を，最初の派生クラスではもちろんのこと，最後の派生クラスでもその仮想関数をオーバーライドできます．たとえば，基本クラスBがf()という仮想関数を含んでいて，D1がBを継承し，さらにD2がD1を継承するとき，D1もD2もそれぞれにおいてf()をオーバーライドできます．

## 例　10.3　仮想関数の詳細

**1.**　次に示すのは，前節の例4に示したプログラムの改良版です．getarea()関数を基本クラスareaで純粋仮想関数として宣言しています．

```
// 抽象クラスを作成する
#include <iostream>
using namespace std;

class area {
  double dim1, dim2; // 図形の寸法
public:
  void setarea(double d1, double d2)
  {
```

```
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea() = 0; // 純粋仮想関数
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

int main()
{
  area *p;
  rectangle r;
  triangle t;
  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);
  p = &r;
  cout << "Rectangle has area: " << p->getarea() << '¥n';

  p = &t;
  cout << "Triangle has area: " << p->getarea() << '¥n';
  return 0;
}
```

getarea()は純粋仮想関数であるため，各派生クラスで必ずオーバーライドされます．

**2.** 次のプログラムを見ると，仮想関数の継承ではその仮想性も保たれることがわかります.

```
// 仮想関数は，継承されても仮想性を保つ
#include <iostream>
using namespace std;

class base {
public:
  virtual void func()
  {
    cout << "Using base version of func()¥n";
  }
};

class derived1 : public base {
public:
  void func()
  {
    cout << "Using derived1's version of func()¥n";
  }
};

// derived2がderived1を継承
class derived2 : public derived1 {
public:
  void func()
  {
    cout << "Using derived2's version of func()¥n";
  }
};

int main()
{
  base *p;
  base ob;
  derived1 d_ob1;
  derived2 d_ob2;

  p = &ob;
  p->func(); // baseのfunc()を使用する

  p = &d_ob1;
  p->func(); // derived1のfunc()を使用する

  p = &d_ob2;
  p->func(); // derived2のfunc()を使用する
```

```
    return 0;
}
```

このプログラムでは，まず，derived1が仮想関数func()を継承し，func()をオーバーライドします．次に，derived2がderived1を継承します．func()は，今度はderived2でオーバーライドされます．

仮想関数は階層を構成するため，仮にderived2がfunc()をオーバーライドしなかった場合，d_ob2がアクセスされると，derived1のfunc()が使用されます．derived1もderived2もfunc()をオーバーライドしなかったとすれば，同関数への参照はすべてbaseで定義されているfunc()に向かいます．

### 練習問題 10.3 仮想関数の詳細

1. 2つのプログラム例を使って，いろいろと試してみなさい．たとえば，例1のareaを使ってオブジェクトを作成し，どのようなエラーメッセージが出力されるかを確認しなさい．例2では，derived2にあるfunc()の再定義を削除し，確かにderived1のバージョンが使用されるか確認しなさい．

2. 抽象クラスを使用してオブジェクトを作成できない理由を説明しなさい．

3. 例2で，derived1のfunc()再定義だけを削除するとどうなるか説明しなさい．これでも，プログラムをコンパイルして実行できるかどうか確認し，できるとすれば，その理由を説明しなさい．

## 10.4 ポリモーフィズムの応用

以上の説明で，仮想関数を使って実行時ポリモーフィズムを実現する方法は理解できたと思います．次に，それをどのように使用するか，また，なぜ使用するかを考えてみましょう．本書で何度も述べてきたとおり，ポリモーフィズムこそ，共通のインターフェイスをいくつもの類似した(しかし，技術的には異なる)状況に応じて，「1つのインターフェイス，複数のメソッド」を実現するための手段です．ポリモーフィズムは，複雑なシステムを大幅に単純化する方法として重要です．巧みに定義された1つのインターフェイスを使えば，異なる(しかし，関連性のある)複数のアクションにアクセスでき，不自然な複雑さが軽減されます．本質的に，ポリモーフィズムは類似のアクション間の論理関係を明白にす

るため，プログラムは理解しやすく，保守しやすいものになります．関連した複数のアクションに共通のインターフェイスでアクセスできれば，覚えることもそれだけ少なくて済みます．

オブジェクト指向プログラミング全般，中でもC++に関してよく使われる用語に，コンパイル時バインディングと実行時バインディングという2つの用語があります．重要な言葉なのでよく理解しておいてください．**コンパイル時バインディング**(early binding)とは，基本的に，コンパイル時にわかっているイベントのことを言います．具体的には，コンパイル時に解決できる関数呼び出しのことです．「通常の」関数，オーバーロードされた関数，仮想性のないメンバ関数とフレンド関数などがコンパイル時にバインドされます．この種の関数のコンパイルでは，それを呼ぶために必要なすべてのアドレス情報が，すでにコンパイルの時点でわかっています．コンパイル時バインディングの最大の利点は，効率の良さです．コンパイル時バインディングが広く使用されているのは，そのためです．コンパイル時にバインドされた関数の呼び出しは，最速の関数呼び出しです．反対に，最大の欠点は柔軟性に欠けることです．

**実行時バインディング**(late binding)とは，実行時にならないとわからないイベントのことを言います．実行時にバインドされる関数呼び出しでは，プログラムが実行されるまで，呼ばれる関数のアドレスがわかりません．実行時にバインドされるオブジェクトとして，C++には仮想関数があります．仮想関数が基本クラスポインタを通じてアクセスされると，プログラムは，いったいどの型のオブジェクトが指し示されているかを実行時になってから判断し，オーバーライドされている関数のどのバージョンを実行するかを選択しなければなりません．実行時バインディングの最大の利点は，実行時における柔軟性です．プログラムはランダムなイベントに自由に対応でき，大量の「不測の事態に備えたコード」を用意する必要がありません．反対に，最大の欠点は関数呼び出しのためのオーバーヘッドが大きいことです．このため，コンパイル時にバインドされた関数呼び出しに比べて，一般には遅くなります．

効率と柔軟性のトレードオフをよく考え，いつコンパイル時バインディングを使用し，いつ実行時バインディングを使用するかを適切に見極めなければなりません．

## 例　10.4　ポリモーフィズムの応用

1. 次に示すのは，「1つのインターフェイス，複数のメソッド」を実感できるプログラム例です．これは，整数値の抽象リストクラスを定義します．リストへのインターフェイスは，store()とretrieve()という純粋仮想関数で定義されます．値を格納するにはstore()関数を呼び，リストから値を取り出すにはretrieve()を呼びます．基本クラスlistには，これらのアクションのデフォルトメソッドが何も定義されていません．その代わり，個々の派生クラスで，どの種のリストを維持するかを定義します．このプログラムでは，キューとスタックという2種類のリストを実装します．この2つのリストはまったく異なる動作をしますが，同じインターフェイスでアクセスされます．

このプログラムをよく見て学習してください.

```
// 仮想関数の実例
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
  list *head; // リスト先頭へのポインタ
  list *tail; // リスト末尾へのポインタ
  list *next; // 次項目へのポインタ
  int num;    // 格納される値
  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// キュー型リストの作成
class queue : public list {
public:
  void store(int i);
  int retrieve();
};

void queue::store(int i)
{
  list *item;
  item = new queue;

  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // リスト末尾に置く
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;
```

```
  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;
  return i;
}

// スタック型リストの作成
class stack : public list {
public:
  void store(int i);
  int retrieve();
};

void stack::store(int i)
{
  list *item;
  item = new stack;

  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // スタックのような操作になるよう，リスト最前部に置く
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
```

```
  p = head;
  head = head->next;
  delete p;
  return i;
}

int main()
{
  list *p;

  // キューのデモ
  queue q_ob;
  p = &q_ob; // キューを指す

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Queue: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '¥n';

  // スタックの実演
  stack s_ob;
  p = &s_ob; // スタックを指す

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Stack: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '¥n';

  return 0;
}
```

**2.** 例 1 で示したプログラムの main() 関数では，単に list クラスが正しく動作することがわかるにすぎません．しかし，代わりに次のmain()を使うと，実行時ポリモーフィズムの強力さが実感できます．

```
int main()
{
  list *p;
  stack s_ob;
  queue q_ob;
  char ch;
  int i;

  for(i=0; i<10; i++) {
    cout << "Stack or Queue? (S/Q): ";
    cin >> ch;
    ch = tolower(ch);
    if(ch=='q') p = &q_ob;
    else p = &s_ob;
    p->store(i);
  }

  cout << "Enter T to terminate¥n";
  for(;;) {
    cout << "Remove from Stack or Queue? (S/Q): ";
    cin >> ch;
    ch = tolower(ch);
    if(ch=='t') break;
    if(ch=='q') p = &q_ob;
    else p = &s_ob;
    cout << p->retrieve() << '¥n';
  }

  cout << '¥n';

  return 0;
}
```

このmain()を見ると，実行時に起こるランダムイベントを，仮想関数と実行時ポリモーフィズムで容易に処理できることがわかります．このプログラムは，forループを0から9まで反復実行します．ループの反復のたびに，どの種類のリスト(スタックまたはキュー)に値を入れるのか選択するよう要求されます．返す答えに応じて，基本ポインタpが正しいオブジェクトに向けて設定され，iの現在の値が格納されます．1つのループが終了すると，次のループが始まり，どちらのリストから値を取り出すかの指示を求めてきます．ここでも，ユーザーからの応答次第で，選択されるリストが決まります．

特に意味のある例ではありませんが，実行時ポリモーフィズム使うことによって，ランダムイベントに応答するプログラムがいかに単純になるかがわかります．たとえば，

Windows オペレーティングシステムは，メッセージを送信することによりプログラムとやり取りします．プログラムにとって，このメッセージはランダムに生成されるものなので，それを受け取ってから初めて応答を判断することになります．そのようなメッセージに応答する方法の１つが，仮想関数の使用です．

### 練習問題　10.4　ポリモーフィズムの応用

1. 例１のプログラムに，もう１つ別の種類のリストを付け加えなさい．新しいリストはソート済みリストで，順序は昇順です．このリストの名前を sorted とします．
2. 実行時ポリモーフィズムを応用することにより，問題に対する解決を単純化できるような方法を考えなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. 仮想関数とは何かを説明しなさい．
2. 仮想関数にすることのできない関数にはどのような種類のものがあるか説明しなさい．
3. 実行時ポリモーフィズムを実現するのに，仮想関数はどのような働きをするか，具体的に答えなさい．
4. 純粋仮想関数とは何かを説明しなさい．
5. 抽象クラスとは何か，ポリモーフィッククラスとは何か，それぞれ説明しなさい．
6. 次のプログラムコードは正しいか，正しくないとすればその理由は何か説明しなさい．

```
class base {
public:
  virtual int f(int a) = 0;
  //  ...
};

class derived : public base {
public:
```

```
    int f(int a, int b) { return a*b; }
    //  ...
};
```

**7.** 仮想性は継承されますか.

**8.** 仮想関数を使っていろいろと試してみなさい. 仮想関数は重要な概念です. この技術を十分にマスターしてください.

## 総合理解度チェック

次の問題を解き, この章で学んだ知識を, 前の章までに学んだ知識に統合できたかどうかを確認しましょう.

**1.** 10.4 節の例 1 で, + 演算子と -- 演算子をオーバーロードしてリスト操作を強化します. + で要素を格納し, -- で要素を取り出すようにしなさい.

**2.** 仮想関数は, オーバーロードされた関数とどう違うか説明しなさい.

**3.** 本書でこれまでに扱ってきた関数オーバーロードの例を見直しなさい. 仮想関数に変換できるものはないか, また, これまでに自分が突き当たったプログラミングに関する問題の中に, 仮想関数で解決できるものはないか考えなさい.

# 11

# テンプレートと例外処理

## この章の内容

この章では，C++の中でも特に高度な2つの機能，テンプレート(template）と例外処理(exception handling)について説明します．これらの2つの機能はどちらもC++の最初の仕様には含まれていませんでしたが，数年前に追加され，標準C++で定義されました．最近のC++コンパイラはどれでもこれらの機能をサポートしています．これらの機能を利用すれば，プログラミングにおける最も困難な目標のうちの2つを達成することができます．2つの目標とは，再利用が可能で，堅牢なプログラムコードを作成するということです．

テンプレートを使用すれば，汎用の関数とクラスを作成することができます．汎用関数と汎用クラスでは，処理するデータの型を仮引数として指定します．これによって，データ型ごとに専用のプログラムコードを明示的に書き直さなくても，1つの関数またはクラスをさまざまな型のデータに対して使用することができます．つまり，テンプレートを使用すれば再利用可能なプログラムコードを作成することができるのです．この章では，汎用関数と汎用クラスについて説明します．

例外処理はC++のサブシステムです．例外処理を使用すれば，実行時に発生したエラーを構造化された方法で処理することができます．C++の例外処理では，エラーが発生したときにエラー処理ルーチンを自動的に呼び出すことができます．例外処理の基本的な利点は，以前であれば大きなプログラムでは必ず「手作業で」記述しておかなければならなかったエラー処理コードの大部分を自動化できるということです．例外処理を正しく利用すれば，堅牢なプログラムを作成することができるのです．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. 仮想関数とは何かを説明しなさい．

2. 純粋仮想関数とは何かを説明しなさい．また，クラス宣言に純粋仮想関数が含まれる場合，そのクラスは何と呼ばれるか，このような使用方法に対する制限は何かを説明しなさい．

3. 実行時ポリモーフィズムは＿＿＿＿＿＿関数と＿＿＿＿＿＿クラスポインタを使用して行います(下線部を埋めなさい)．

4. クラス階層内で派生クラスで，派生クラスが(非純粋)仮想関数の上書きを拒否した場合，その派生クラスのオブジェクトがその関数を呼び出すと，何が起こるか説明しなさい．

5. 実行時ポリモーフィズムの主な利点について説明しなさい．また，欠点についても説明しなさい．

# 11.1　汎用関数

**汎用関数**(generic function)とは，さまざまな型のデータに適用できる一連の汎用操作を定義する関数のことです．汎用関数は，処理するデータの型を仮引数として受け取ります．このしくみを用いれば，1つの汎用プロシージャを広範囲なデータに対して使用することができます．ご存じのとおり，どのような型のデータを扱う際にも，多くのアルゴリズムの論理は同じです．たとえばクイックソートのアルゴリズムは，整数の配列に適用する場合でも，float型の配列に適用する場合でも同じです．異なるのは整列するデータの型だけです．汎用関数を作成すれば，データの型にとらわれることなく，本質的なアルゴリズムを定義することができます．汎用関数を使用すれば，関数の実行時に実際に処理するデータの型に対して適切なプログラムコードが，コンパイラによって自動的に生成されます．つまり，汎用関数を作成すると，自動的に自分自身をオーバーロードすることのできる関数を作成したことになるのです．

汎用関数を作成するには，templateキーワードを使用します．template(テンプレート)という言葉は，C++におけるこのキーワードの役割をそのまま表しています．このキーワードを使うと，その関数の動作内容を示すテンプレート(フレームワーク)を作成し，詳細については，必要に応じてコンパイラに補足させることができます．テンプレート関数定義の一般形式を次に示します．

**テンプレート関数定義**

```
template <class Ttype> ret-type func-name( parameter list )
{
    // 関数の本体
}
```

*Ttype*は，関数で使用するデータ型を指定するためのプレースホルダです．この名前を関数定義内で使用することができます．ただし，これは単なるプレースホルダであり，コンパイラは各データ型に専用の関数を作成する際に，これを実際のデータ型と置き換えます．

テンプレート宣言内ではclassキーワードを使用して汎用型を指定するのが一般的ですが，typenameキーワードを使うこともできます．

## 例 11.1 汎用関数

**1.** 次のプログラムでは，渡された2つの変数の値を入れ替える汎用関数を使用しています．「2つの値を入れ替える」という汎用処理は，変数の型とは無関係なので，この処理は汎用関数として作成するのが適しています．

```
// 汎用テンプレートの例
#include <iostream>
using namespace std;

// 関数テンプレート
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // 整数を入れ替える
  swapargs(x, y); // floatを入れ替える

  cout << "Swapped i, j: " << i << ' ' << j << endl;
  cout << "Swapped x, y: " << x << ' ' << y << endl;

  return 0;
}
```

このプログラムでは，templateキーワードを使用して汎用関数を定義しています．次の行を見てみましょう．

```
template <class X> void swapargs(X &a, X &b)
```

この行では，テンプレートを作成することと，汎用定義がここから始まるという2つのことをコンパイラに伝えています．Xはプレースホルダとして使用する汎用型です．template部分の後にはswapargs()関数の宣言が続いており，入れ替える値のデータ

型としてXを使用しています．main()関数内では，整数とfloatの2種類のデータ型を使用して，swapargs()関数を呼び出しています．swapargs()関数は汎用関数なので，コンパイラは2つのswapargs()関数を自動的に作成します．1つは整数値を入れ替えるswapargs()関数，もう1つはfloat型の値を入れ替えるswapargs()関数です．さっそくこのプログラムをコンパイルして試してみてください．

ここで，テンプレートについて説明する際に使われるその他の用語について説明しておきましょう．これらの用語はほかのC++の資料でも見かけることがあるでしょう．まず，汎用関数(つまり，先頭にtemplateキーワードの付いた関数定義)は**テンプレート関数**(template function) とも呼ばれます．コンパイラがこの関数を各データ型専用に作成する際，その関数は**生成された関数**(generated function)と呼ばれます．関数を生成するという行為は，関数を**インスタンシエートする**(instantiate)と呼ばれます．換言すれば，生成された関数はテンプレート関数の1つのインスタンスであると言えます．

**2.** 汎用関数定義のtemplate部分は，関数名と同じ行に記述する必要はありません．たとえば，次のようにしてswapargs()関数を記述することもできます．

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

この形式を使う場合は，template文と汎用関数定義の開始行の間にほかの文を挿入しないように注意してください．たとえば，次のプログラムコードはコンパイルできません．

```
// このプログラムコードはコンパイルできない
template <class X>
int i; // これはエラー
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

コメントに示したとおり，template文は関数定義の直前に置かなければなりません．

3. 前述のとおり，テンプレート定義内ではclassキーワードを使う代わりに，typenameキーワードを使用して汎用型を指定することもできます．次に，typenameキーワードを使用してswapargs()関数を定義する例を示します．

```
// typenameキーワードを使用する
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

typenameキーワードを使って，テンプレート内で未知の型を指定することもできますが，本書ではこの方法については説明しません．

4. template文でカンマ区切りのリストを使用すれば，汎用データ型を複数定義することもできます．次のプログラムでは，2つの汎用型を使用して汎用関数を作成しています．

```
#include <iostream>
using namespace std;

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << endl;
}

int main()
{
  myfunc(10, "hi");
  myfunc(0.23, 10L);

  return 0;
}
```

この例では，プレースホルダ型のtype1とtype2は，コンパイラがmyfunc()関数のインスタンスを作成する際に，それぞれint型とchar * 型，double型とlong型に置き換えられます．

**重要** 汎用関数を作成すると，コンパイラはプログラム内からその関数が呼び出される各種の方法に対処するために，その関数のインスタンスを必要なだけいくつでも作成できることになります．

**5.** 汎用関数はオーバーロード関数と似ていますが，汎用関数の方が制限が厳しくなっています．関数をオーバーロードする際には，各関数ごとにさまざまな種類の処理を実行することができます．しかし，汎用関数ではすべての関数内で同じ汎用処理を実行しなければなりません．たとえば次のオーバーロード関数は，それぞれ異なる処理を行っているので，汎用関数に置き換えることはできません．

```
void outdata(int i)
{
  cout << i;
}

void outdata(double d)
{
  cout << setprecision(10) << setfill('#');
  cout << d;
  cout << setprecision(6) << setfill(' ');
}
```

**6.** テンプレート関数は必要に応じて自分自身をオーバーロードしますが，テンプレート関数を明示的にオーバーロードすることもできます．汎用関数をオーバーロードすると，その関数に対しては，汎用関数がオーバーロード関数によってオーバーライドされ(隠され)ます．次のプログラムを見てみましょう．これは例1のプログラムを修正したものです．

```
// テンプレート関数を上書きする
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

// これによって汎用のswapargs()関数が上書きされる
void swapargs(int a, int b)
{
  cout << "this is inside swapargs(int,int)¥n";
}

int main()
{
```

```
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // オーバーロードされたswapargs()関数を呼び出す
  swapargs(x, y); // floatを入れ替える

  cout << "Swapped i, j: " << i << ' ' << j << endl;
  cout << "Swapped x, y: " << x << ' ' << y << endl;

  return 0;
}
```

コメントに示したとおり，swapargs(i, j)関数を呼び出すと，プログラム内で定義されている明示的なオーバーロード関数swapargs()が呼び出されます．汎用関数は明示的なオーバーロードによって上書きされているので，コンパイラはこのデータ型用の汎用swapargs()関数を生成しません．

この例に示したように，テンプレートを手作業でオーバーロードすると，特別な状況に合わせて汎用関数の特定のインスタンスを調整することができます．ただし，データ型によって異なる関数が必要なときは，テンプレートよりもオーバーロード関数を使用する方が一般的です．

## 練習問題　11.1　汎用関数

1. 前述の各プログラムをまだ実行していない場合は，ここで試しなさい．

2. min()という名前の汎用関数を作成し，2つの引数のうち小さい方を返しなさい．たとえば，min(3, 4)は3を返し，min('c', 'a')はaを返すようにします．また，作成した関数をプログラムに組み込んで動作を試しなさい．

3. テンプレート関数の好例として，find()という関数があります．この関数は，配列内であるオブジェクトを検索します．オブジェクトが見つかった場合はそのオブジェクトのインデックスを返し，オブジェクトが見つからなかった場合は-1を返します．次に，特定のデータ型用のfind()関数のプロトタイプを示します．このfind()関数を修正して汎用関数を作成し，プログラム内でその動作を確認しなさい(size仮引数には，配列内の要素数を指定します)．

```
int find(int object, int *list, int size)
{
  // ...
}
```

**4.** 汎用関数の利点と，汎用関数がソースコードを簡略化するのに役立つ理由を，自分の言葉で説明しなさい．

# 11.2　汎用クラス

汎用関数に加えて，汎用クラス(generic class)を定義することもできます．汎用クラスを定義する際には，そのクラスで使用するすべてのアルゴリズムを定義したクラスを作成しますが，操作する実際のデータの型は，そのクラスのオブジェクトを作成する際に仮引数として指定します．

汎用クラスは，クラスに汎用化できる論理が含まれているときに便利です．たとえば，整数のキューを管理するのと同じアルゴリズムを使用して，文字のキューを処理することもできます．また，メールアドレスのリンクリストを管理するのと同じしくみを使用して，自動車部品情報のリンクリストを処理することができます．汎用クラスを使用すれば，すべてのデータ型のキューやリンクリストを管理するクラスを作成することができます．コンパイラは，オブジェクトの作成時に指定された型に応じて，適切な型のオブジェクトを自動的に生成します．

汎用クラス宣言の一般形式を次に示します．

**汎用クラスの宣言**

```
template <class Ttype> class class-name {
    .
    .
    .
}
```

*Ttype*はクラスをインスタンシエートする際に指定される型名のプレースホルダです．必要であれば，カンマで区切ったリストを使用して，複数の汎用データ型を定義することもできます．

汎用クラスを作成したら，次の一般形式を使用して，そのクラスの特定のインスタンスを作成することができます．

**インスタンスの作成**

```
class-name <type> ob;
```

*type*にはそのクラスで操作するデータの型名を指定します．

汎用クラスのメンバ関数は自動的に汎用関数となります．templateキーワードを明示的に指定する必要はありません．

第14章で説明しますが，C++ではテンプレートクラスのライブラリが組み込まれています．このライブラリは，一般に**標準テンプレートライブラリ**（Standard Template Library：STL）と呼ばれています．このライブラリには，広く使われるアルゴリズムおよびデータ構造体の汎用バージョンが含まれています．STLを効率的に使うためには，テンプレートクラスとその構文について十分に理解しておかなければなりません．

## 例 11.2　汎用クラス

**1.** このプログラムでは，ごく単純な汎用リンクリストクラスを作成しています．そして，文字を格納するリンクリストを作成し，このクラスの動作を確認しています．

```
// 単純な汎用リンクリスト
#include <iostream>
using namespace std;

template <class data_t> class list {
  data_t data;
  list *next;
public:
  list(data_t d);
  void add(list *node) {node->next = this; next = 0; }
  list *getnext() { return next; }
  data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
  data = d;
  next = 0;
}

int main()
{
  list<char> start('a');
  list<char> *p, *last;
  int i;

  // リストを作成する
  last = &start;
```

```
    for(i=1; i<26; i++) {
      p = new list<char> ('a' + i);
      p->add(last);
      last = p;
    }

    // リストを追跡する
    p = &start;

    while(p) {
      cout << p->getdata();
      p = p->getnext();
    }

    return 0;
  }
```

ご覧のとおり，汎用クラスの宣言は汎用関数の宣言と似ています．リストに実際に格納されるデータの型は，クラス宣言内では汎用になっています．実際のデータ型が決まるのは，リストのオブジェクトを実際に宣言するときです．この例では，main()関数内でオブジェクトとポインタを作成し，リストのデータ型をcharと指定しています．この宣言部分に注目してみましょう．

```
list<char> start('a');
```

データ型を山括弧(<>)で囲んで指定しています．

このプログラムを実際に入力し，実行してみてください．このプログラムでは，アルファベットを含むリンクリストを作成し，それを表示しています．ただし，listオブジェクトの作成時に指定するデータ型を変えるだけで，リストに保存するデータの型を変更することができます．たとえば，次の宣言を使用すれば，整数を格納するほかのオブジェクトを作成することができます．

```
list<int> int_start(1);
```

また，自分で作成したデータ型をリストに保存することもできます．たとえば，アドレス情報を保存したい場合は，次の構造体を使用します．

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
}
```

listクラスを使用して，addr型オブジェクトを保存するオブジェクトを生成するには，次のような宣言を使用します(ここでは，structvarに有効なaddr構造体が含まれているものとします).

```
list<addr> obj(structvar);
```

**2.** 次に示すのは，もう1つの汎用クラスの例です．これは第1章で紹介したstackクラスを修正したものです．ここではstackクラスをテンプレートクラスとして使用しています．したがって，stackクラスにはどの型のオブジェクトでも格納できます．この例では，文字スタックと浮動少数点数スタックを作成しています．

```
// 汎用stackクラスの使用例
#include <iostream>
using namespace std;

#define SIZE 10

// 汎用stackクラスを作成する
template <class StackType> class stack {
  StackType stck[SIZE];     // スタック領域を確保する
  int tos;                  // スタックの先頭の索引
public:
  void init() { tos = 0; } // スタックを初期化する
  void push(StackType ch); // スタックにオブジェクトをプッシュする
  StackType pop();          // スタックからオブジェクトをポップする
};

// オブジェクトをプッシュする
template <class StackType>
void stack<StackType>:: push(StackType ob)
{
  if(tos==SIZE) {
    cout << "Stack is full.¥n";
    return;
  }
  stck[tos] = ob;
  tos++;
}

// オブジェクトをポップする
template <class StackType>
StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "Stack is empty.¥n";
    return 0; // スタックが空の場合はヌルを返す
```

```
  }
  tos--;
  return stck[tos];
}

int main()
{
  // 文字スタックの動作を確認する
  stack<char> s1, s2;  // 2つのスタックを作成する
  int i;

  // スタックを初期化する
  s1.init();
  s2.init();
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "¥n";

  // double型スタックの動作を確認する
  stack<double> ds1, ds2;  // 2つのスタックを作成する

  // スタックを初期化する
  ds1.init();
  ds2.init();

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop()
  ➥<< "¥n";
  for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop()
  ➥<< "¥n";

  return 0;
}
```

このstackクラス(および前述のlistクラス)からわかるとおり，汎用関数と汎用クラスは非常に強力な機能です．これらを利用すれば，すべてのデータ型で使用できる汎

用のアルゴリズムを定義できるので，プログラミングの効率を最大限に高めることができます．アルゴリズムを適用したいデータ型ごとにプログラムコードを作成するという退屈な作業から解放されます．

**3.** テンプレートクラスには，複数の汎用データ型を定義することができます．このためには，そのクラスで使用するすべてのデータ型をテンプレート宣言内でカンマで区切って宣言するだけです．次の短いプログラムでは，2つの汎用データ型を使用するクラスを作成しています．

```
/* このプログラムでは，1つのクラス定義内で
   2つの汎用データ型を使用する
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '¥n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "This is a test");

  ob1.show(); // int, doubleを表示する
  ob2.show(); // char, char *を表示する

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
10 0.23
X This is a test
```

このプログラムでは，2つの型のオブジェクトを宣言しています．ob1では整数とdouble型のデータを使用します．ob2では文字と文字ポインタを使用します．どちらの場合も，オブジェクトの作成方法に合わせて，コンパイラが適切なデータと関数を自動的に生成します．

**練習問題 11.2 汎用クラス**

1. 前述の2つの汎用クラスの例をまだ実際に試していない場合は，ここでコンパイルし，実行しなさい．また，データ型の異なるリストとスタックを宣言しなさい．

2. 汎用キュークラスを作成し，その動作を確認しなさい．

3. input という名前の汎用クラスを作成し，コンストラクタが呼び出されたときに次の処理を実行しなさい．

   - プロンプトを表示し，ユーザーに入力を求める
   - ユーザーが入力したデータを入力する
   - データが定義済みの範囲内に収まらない場合は，再びプロンプトを出す

   input 型のオブジェクトは，次のように宣言します．

   ```
   input ob("prompt message", min-value, max-value)
   ```

   *prompt message*には，入力を求めるプロンプトメッセージを指定します．*min-value*と*max-value*には，それぞれ受け付ける最小値と最大値を指定します(ユーザーが入力するデータの型は，*min-value* および *max-value* の型と同じです)．

# 11.3 例外処理

C++ には，**例外処理**(exception handling) というエラー処理機構が組み込まれています．例外処理を使うと，実行時エラーをより容易に管理し，対処することができます．C++ の例外処理では，try，catch，throw の3つのキーワードが基盤となっています．最も一般的な形では，例外が発生したかどうかを監視したいプログラム文を try ブロック内に含めます．try ブロック内で例外(エラー)が発生すると，その例外は throw 文を使用して投げられ，catch 文を使用して捕獲されて処理されます．

前述のとおり，例外を投げる文は，すべてtryブロック内から実行する必要があります(tryブロック内から呼び出した関数でも例外を投げることができます)．すべての例外は，例外を投げたtry文の直後にある catch 文によって捕獲する必要があります．次に，try 文と catch 文の一般形式を示します．

try文とcatch文

```
try {
   // tryブロック
}
catch (type1 arg) {
   // catchブロック
}
catch (type2 arg) {
   // catchブロック
}
catch (type3 arg) {
   // catchブロック
}
  .
  .
  .
catch (typeN arg) {
   // catchブロック
}
```

tryブロックには，例外を監視する部分のプログラムを含めます．tryブロックには，1つの関数内の数行の文を含めても，またmain()関数のプログラムコードをすべて含めてもかまいません(この場合，実質的にはプログラム全体を監視することになります)．

例外が投げられると，対応するcatch文によって捕獲され，例外が処理されます．1つのtryブロックに複数のcatch文を対応付けることができます．その場合，どのcatch文を使用するかは例外の型によって決まります．つまり，例外の型と一致する型を指定しているcatch文が実行されます(残りのcatch文はすべて無視されます)．例外を捕獲すると，argはその値を受け取ります．例外自体にアクセスする必要がない場合は，catch句で型だけを指定することもできます．argは任意指定です．自分で作成したクラスを含め，すべての型のデータを捕獲することができます．実際に，クラス型が例外として使われることはよくあります．

throw文の一般形式を次に示します．

throw文

```
throw exception;
```

throwは，tryブロック内から実行するか，ブロック内のプログラムコードから(直接的または間接的に)呼び出す関数内で実行します．*exception*には，投げる値を指定します．

対応するcatch文のない例外を投げると，プログラムが異常終了します．標準C++に準拠したコンパイラでは，処理されない例外が投げられると，標準ライブラリ関数であるterminate()関数が呼び出されます．デフォルトでは，terminate()関数はabort()関数を呼び出してプログラムを停止しますが，必要であれば，独自の終了ハンドラを定義することもできます．詳細については，使用するコンパイラのライブラリリファレンスを参照してください．

## 例 11.3　例外処理

**1.**　次のプログラムは，C++の例外処理のしくみを示す単純な例です．

```
// 単純な例外処理の例
#include <iostream>
using namespace std;

int main()
{
  cout << "start¥n";

  try { // tryブロックの開始
    cout << "Inside try block¥n";
    throw 10; // エラーを投げる
    cout << "This will not execute";
  }

  catch (int i) { // エラーを捕獲する
    cout << "Caught One! Number is: ";
    cout << i << "¥n";
  }

  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
start
Inside try block
Caught One! Number is: 10
end
```

このプログラムについて詳しく見てみましょう．このプログラムには，3つの文を含むtryブロックと，整数例外を処理する1つのcatch(int i)文があります．tryブロック内

では，3つのうち2つの文(1つ目のcout文とthrow文)だけが実行されます．例外が投げられると，制御はcatch式に移り，tryブロックは終了します．つまり，catchは呼び出されるのではなく，プログラムの制御がcatchに移動します(このために，スタックは自動的に再設定されます)．したがって，throwの後に続くcout文は実行されません．

catch文の実行後，プログラムの制御はcatchの後に続く文に移ります．ただし，例外処理の主な目的は破壊的なエラーを処理することですから，catchブロックの最後にはexit()関数やabort()関数，あるいはプログラムを終了するその他の関数を実行することがほとんどです．

**2.** 前述のとおり，例外の型は，catch文内で指定した型に一致しなければなりません．たとえば，前述のプログラムでは，catch文内の型をdoubleに変更すると，例外は捕獲されず，異常終了が行われます．このように修正したプログラムを次に示します．

```
// このプログラムは正しく動作しない
#include <iostream>
using namespace std;

int main()
{
  cout << "start¥n";

  try { // tryブロックの開始
    cout << "Inside try block¥n";
    throw 10; // エラーを投げる
    cout << "This will not execute";
  }

  catch (double i) { // 整数例外を処理しない
    cout << "Caught One! Number is: ";
    cout << i << "¥n";
  }

  cout << "end";

  return 0;
}
```

整数例外はdouble catch文によって捕獲されないため，このプログラムからの出力は次のようになります．

```
start
Inside try block
Abnormal program termination
```

**3.** tryブロック内から呼び出される関数にthrow文を記述すれば，tryブロック外の文からでも例外を投げることができます．たとえば，次のようなプログラムを作成することができます．

```
/* tryブロック外の関数から
   例外を投げる
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Inside Xtest, test is: " << test << "¥n";
  if(test) throw test;
}

int main()
{
  cout << "start¥n";

  try { // tryブロックの開始
    cout << "Inside try block¥n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }

  catch (int i) { // エラーを捕獲する
    cout << "Caught One! Number is: ";
    cout << i << "¥n";
  }

  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught One! Number is: 1
end
```

**4.** 1つの関数に対してローカルなtryブロックを作成することができます．この場合，その関数を実行するたびに，その関数に対して相対的な例外処理が再設定されます．次のプログラムを見てみましょう．

```
#include <iostream>
using namespace std;

// main()以外の関数にtry/catch文を含めることができる
void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '¥n';
  }
}

int main()
{
  cout << "start¥n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
start
Caught One!  Ex. #: 1
Caught One!  Ex. #: 2
Caught One!  Ex. #: 3
end
```

ご覧のとおり，3つの例外が投げられています．各例外の後，関数は返ります．この関数を再び呼び出すときには，例外処理が再設定されます．

**5.** 前述のとおり，1つのtry文に複数のcatch文を関連付けることができます．実際にはそうすることが極めて一般的です．ただし，各catch文では，型の異なる例外を捕獲しなければなりません．たとえば，次のプログラムでは整数と文字列を捕獲しています．

```
#include <iostream>
using namespace std;

// 異なる型の例外を捕獲できる
void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "Value is zero";
  }

  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '¥n';
  }

  catch(char *str) {
    cout << "Caught a string: ";
    cout << str << '¥n';
  }
}

int main()
{
  cout << "start¥n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
start
Caught One!  Ex. #: 1
Caught One!  Ex. #: 2
Caught a string: Value is zero
Caught One!  Ex. #: 3
end
```

ご覧のとおり，各 catch 文は特定の型にしか対処していません．

一般に，catch 文はプログラム内に記述した順に検査されます．そして，一致する文が 1 つだけ実行されます．残りの catch ブロックはすべて無視されます．

## 練習問題 11.3 例外処理

1. C++例外処理のしくみを理解する最良の方法は，実際に試してみることです．前述のサンプルプログラムを入力し，コンパイルして実行しなさい．次に，これらを部分的に修正して試し，結果を観察しなさい．

2. 次のプログラムコードの誤りを指摘しなさい．

```
int main()
{
  throw 12.23;
```

3. 次のプログラムコードの誤りを指摘しなさい．

```
try {
  // ...
  throw 'a';
  // ...
}

catch(char *) {
  // ...
}
```

4. 対応するcatch文のない例外を投げると，何が起こるかを説明しなさい．

# 11.4　例外処理の詳細

C++の例外処理をより使いやすくするために，あといくつかの機能について説明しなければなりません．

状況によっては，例外ハンドラを使用して，特定の型だけではなくすべての例外を捕獲したい場合があります．次のcatch構文を使用すれば，これを単純に行うことができます．

catch文

```
catch(...) {
    // すべての例外を処理する
}
```

この構文の省略記号はすべてのデータ型に一致します．

また，関数が呼び出し元に投げて返す例外の型を限定することができます．別の言い方をすれば，1つの関数が外部に対して投げることのできる例外の型を指定することができます．実際には，関数がどのような例外も投げられないようにすることもできます．これらの制限を使用するには，関数定義にthrow句を追加しなければなりません．この一般形式を次に示します．

**throw句**

```
ret-type func-name(arg-list) throw(type-list)
{
   // ...
}
```

この構文を使用すると，この関数は，*type-list*に指定したカンマ区切りリストに含まれるデータ型しか投げることができなくなります．ほかの型の例外を投げると，プログラムが異常終了します．関数が例外をまったく投げられないようにするには，空のリストを指定します．

標準C++に準拠したコンパイラを使用している場合は，関数が許可されていない例外を投げようとすると，標準ライブラリ関数のunexpected()が呼び出されます．デフォルトでは，unexpected()によってterminate()関数が呼び出され，プログラムが異常終了します．ただし，必要であれば独自の終了ハンドラを記述することもできます．独自の終了ハンドラの記述方法については，ご使用のコンパイラのマニュアルを参照してください．

例外ハンドラの内部からさらに例外を投げるには，通常どおりにthrow文を呼び出し，例外を指定しません．すると，現在の例外が外側のtry/catchブロックに渡されます．

## 例 11.4　例外処理の詳細

**1.** 次のプログラムは，catch(...)の使用方法を示しています．

```
// すべての例外を捕獲する
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test;    // intを投げる
    if(test==1) throw 'a';     // charを投げる
    if(test==2) throw 123.23; // doubleを投げる
  }
```

```
  catch(...) { // すべての例外を捕獲する
    cout << "Caught One!¥n";
  }
}

int main()
{
  cout << "start¥n";
  Xhandler(0);
  Xhandler(1);
  Xhandler(2);
  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります.

```
start
Caught One!
Caught One!
Caught One!
end
```

ご覧のとおり，3 つすべての throw 文が 1 つの catch 文によって捕獲されています.

**2.** catch(...)の優れた用途としては，一連の catch 文の最後に使用する方法があります. これによって，便利なデフォルトの「すべてを捕獲する」文を作成することができます. 次のプログラムは，前述のプログラムを修正し，整数例外を明示的に捕獲し，その他の型については catch(...) 文によって捕獲するようにしたものです.

```
// catch(...)をデフォルトとして使用する
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test;    // intを投げる
    if(test==1) throw 'a';     // charを投げる
    if(test==2) throw 123.23; // doubleを投げる
  }

  catch(int i) { // int例外を捕獲する
    cout << "Caught " << i << '¥n';
  }
```

```
    catch(...) { // その他のすべての例外を捕獲する
      cout << "Caught One!¥n";
    }
  }

  int main()
  {
    cout << "start¥n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "end";

    return 0;
  }
```

このプログラムからの出力を次に示します.

```
start
Caught 0
Caught One!
Caught One!
end
```

このプログラムからわかるように，catch(...)をデフォルトとして使用すれば，明示的に処理する必要のないすべての例外を捕獲することができます．また，すべての例外を捕獲すると，処理されない例外によってプログラムが異常終了されるのを防ぐことができます.

**3.** 次のプログラムは，1つの関数から投げることができる例外の型を限定する方法を示しています.

```
// 関数が投げる型を限定する
#include <iostream>
using namespace std;

// この関数は，int，char，doubleしか投げることができない
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test;    // intを投げる
  if(test==1) throw 'a';     // charを投げる
  if(test==2) throw 123.23; // doubleを投げる
}
```

```
int main()
{
  cout << "start¥n";
  try{
    Xhandler(0); // Xhandler()に1と2も渡して実験すること
  }

  catch(int i) {
    cout << "Caught int¥n";
  }

  catch(char c) {
    cout << "Caught char¥n";
  }

  catch(double d) {
    cout << "Caught double¥n";
  }

  cout << "end";

  return 0;
}
```

このプログラムのXhandler()関数は，整数，文字，doubleの例外しか投げることができません．ほかの型の例外を投げようとすると，プログラムが異常終了します(つまり，unexpected()関数が呼び出されます)．試しに，リストからintを削除してプログラムを実行してみてください．

制限を課すことができるのは，関数が呼び出し元のtryブロックに対して返すことのできる例外の型だけだということを理解しておいてください．つまり，その関数内で捕獲するのであれば，関数内のtryブロックからは，どの型の例外でも投げることができます．制限が課されるのは，関数外に例外を投げる場合だけです．

**4.** Xhandler()関数を次のように変更すると，この関数は例外をいっさい投げられなくなります．

```
// この関数は例外を投げられない
void Xhandler(int test) throw()
{
  /* 次の文は動作しなくなる
     これらの文によって，
     プログラムの異常終了が発生する */
  if(test==0) throw test;
  if(test==1) throw 'a';
```

```
    if(test==2) throw 123.23;
  }
```

**5.** 前述のとおり，例外を再度投げることもできます．例外を再度投げる主な理由としては，複数のハンドラで例外を処理できるようにすることが挙げられます．たとえば，1つの例外ハンドラで例外のある側面を処理し，2つ目の例外ハンドラで別の側面を処理したい場合などです．例外は，catchブロック(またはそのブロック内から呼び出した関数)内からしか再び投げることができません．再度投げられた例外は，同じcatch文によって再び捕獲されることはなく，外側のcatch文に進みます．次のプログラムは，例外を再度投げる例を示しています．このプログラムでは，char *例外が再び投げられています．

```
// 例外を再度投げる例
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "hello"; // char *を投げる
  }
  catch(char *) { // char *を捕獲する
    cout << "Caught char * inside Xhandler¥n";
    throw ; // char *を関数外に再度投げる
  }
}

int main()
{
  cout << "start¥n";

  try{
    Xhandler();
  }
  catch(char *) {
    cout << "Caught char * inside main¥n";
  }

  cout << "end";

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
start
Caught char * inside Xhandler
Caught char * inside main
end
```

**練習問題　11.4　例外処理の詳細**

1. 次に進む前に，この節で紹介したサンプルプログラムをすべてコンパイルし，実行しなさい．各プログラムによって生成される出力を確実に理解しなさい．

2. 次のプログラムコードの誤りを指摘しなさい．

```
try {
  // ...
  throw 10;
}
catch(int *p) {
  // ...
}
```

3. 前問のプログラムコードの修正方法を示しなさい．

4. すべての型の例外を捕獲するcatch文を示しなさい．

5. 次に示すのは，divide()という関数のスケルトンです．

```
double divide(double a, double b)
{
  // エラー処理を追加する
  return a/b;
}
```

この関数は，aをbによって除算した結果を返します．C++の例外処理を使用して，この関数に例外ハンドラを追加し，ゼロによる除算エラーが発生するのを防ぎなさい．また，この例外ハンドラの動作を確認しなさい．

# 11.5　new演算子の例外処理

第4章で説明したとおり，最新の仕様では，new演算子による割り当て要求が失敗した場合に例外が投げられます．第4章では例外について説明していなかったので，例外の処理方法については触れませんでした．ここで，new演算子が失敗したときに何が起こるかを正確に学ぶことにしましょう．

本題に入る前に断っておきますが，この節で説明するnew演算子の動作は，標準C++で定義されているものです．第4章で説明したとおり，new演算子が失敗したときの正確な動作は，C++が登場して以来，何度か変更されてきました．C++が最初に登場したときは，new演算子は失敗したときにヌルを返していました．その後変更が加えられて，new演算子が失敗すると例外が発生するようになりました．また，この例外の名前も時を経て変わってきています．最終的には，new演算子が失敗したときはデフォルトで例外を生成し，オプションとしてヌルポインタを返すこともできるようになりました．つまり，コンパイラの製造元や作成時期によって，new演算子は異なる方法で実装されてきました．最終的にはすべてのコンパイラが標準C++に合わせてnew演算子を実装するものと思われますが，現在のところはそうでないコンパイラもあります．この節で紹介するサンプルプログラムをコンパイルできない場合は，コンパイラのマニュアルでnew演算子の実装方法の詳細を調べてください．

標準C++では，割り当て要求が失敗すると，new演算子はbad_alloc例外を投げます．この例外を捕獲しないと，プログラムは終了します．短いサンプルプログラムならばそれでもかまいませんが，実際にアプリケーションを作成する場合には，この例外を捕獲して，適切な方法で対処しなければなりません．この例外にアクセスするには，プログラムに<new>ヘッダをインクルードする必要があります．

メモ

bad_alloc例外は，最初はxallocという名前でした．本書の執筆時点では，古い名前を使用しているコンパイラがまだ数多くあります．しかし，標準C++ではbad_allocという名前が指定されているため，将来的にはこの名前が使われるようになるでしょう．

標準C++では，new演算子による割り当てが失敗したときに，例外を投げる代わりにヌルを返すこともできます．この形式のnew演算子は，最近のC++コンパイラを使用して古いプログラムコードをコンパイルするときに最も役立ちます．また，malloc()関数をnew演算子に置き換える場合にも役立ちます．次に，このnew演算子の構文を示します．

new 演算子

```
p_var = new(nothrow) type;
```

p_varは，*type*型のポインタ変数です．nothrow形式のnew演算子の動作は，数年前に使われていた最初の形式のnewとよく似ています．失敗したときにヌルを返すため，古いプログラムコード内で使用することができ，例外ハンドラを追加する必要がありません．ただし，新しいプログラムコードを作成する際には，例外を生成して処理する方法をとった方がよいでしょう．

## 例 11.5　new 演算子の例外処理

**1.** 次のプログラムでは，new演算子に対してtry/catchブロックを使用し，割り当てエラーを監視しています．

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p;
  try {
    p = new int; // intにメモリを割り当てる
  } catch (bad_alloc xa) {
    cout << "Allocation failure.¥n";
    return 1;
  }

  for(*p = 0; *p < 10; (*p)++)
    cout << *p << " ";

  delete p; // メモリを解放する

  return 0;
}
```

このプログラムで発生した割り当てエラーは，catch 文によって捕獲することができます．

**2.** 例1のプログラムは，通常の環境ではめったに失敗しません．次に示すプログラムでは，割り当てエラーを強制的に発生させることによって，new 演算子の例外投入機能を試しています．このプログラムでは，メモリが不足するまで割り当てを続けています．

```
// 割り当てエラーを強制的に発生させる
#include <iostream>
#include <new>
using namespace std;

int main()
{
  double *p;

  // 最終的にメモリが不足する
```

```
  do {
    try {
      p = new double[100000];
    } catch (bad_alloc xa) {
      cout << "Allocation failure.¥n";
      return 1;
    }
    cout << "Allocation OK.¥n";
  } while(p);

  return 0;
}
```

**3.** 次のプログラムでは，new(nothrow)オプションの使用方法を示しています．これは例2のプログラムを修正したもので，割り当てエラーを強制的に発生させています．

```
// new(nothrow)オプションの使用例
#include <iostream>
#include <new>
using namespace std;

int main()
{
  double *p;

  // 最終的にメモリが不足する
  do {
    p = new(nothrow) double[100000];
    if(p) cout << "Allocation OK.¥n";
    else cout << "Allocation Error.¥n";
  } while(p);

  return 0;
}
```

このプログラムで行っているように，nothrowオプションを使用した場合は，それぞれのメモリ割り当て要求ごとに，new演算子から返されるポインタを調べなければなりません．

## 練習問題　11.5　new 演算子の例外処理

**1.** メモリ割り当てエラーが発生した場合の，newとnew(nothrow)の動作の違いを説明しなさい．

**2.** 新しいC++スタイルを使用して，次のプログラムコードを2とおりの方法で修正しなさい．

```
p = malloc(sizeof(int));

if(!p) {
  cout << "Allocation error.¥n";
  exit(1);
}
```

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** 値の配列のモードを返す汎用関数を作成しなさい．

ヒント　「モード」とは，現れる頻度が最も高い値のことです．

**2.** 値の配列の総計を返す汎用関数を作成しなさい．

**3.** 汎用のバブルソートを作成しなさい(または，ほかの任意のソートアルゴリズムを使用しなさい)．

**4.** stackクラスを修正し，型の異なる2つのオブジェクトをスタックに格納できるようにしなさい．また，修正したクラスの動作を確認しなさい．

**5.** try，catch，throwの一般形式を示しなさい．また，それぞれの役割について自分の言葉で説明しなさい．

**6.** stackクラスを再び修正し，スタックのオーバーフローとアンダーフローを例外として処理しなさい．

**7.** コンパイラのマニュアルを参照し，terminate()関数とunexpected()関数がサポートされているかどうかを調べなさい．一般に，これらの関数から任意の関数を呼び出すように構成することができます．使用するコンパイラでこの機能がサポートされている場合は，処理されない例外に対処する独自の終了関数を作成しなさい．

**8.** new演算子が失敗したときに，ヌルを返す方法よりも，例外を生成する方法の方が優れている理由を考えなさい．

# 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

1. 第6章6.7節の例3で安全配列クラスを紹介しました．このクラスを汎用安全配列に書き換えなさい．

2. 第1章では，abs()関数をオーバーロードしました．すべての数値オブジェクトの絶対値を返す汎用のabs()関数を自分で作成しなさい．

# 12

# 実行時型情報とキャスト演算子

## この章の内容

この章では，C++ 言語に最近追加された２つの機能，**実行時型情報**(run-time type identification：RTTI)と新しい**キャスト演算子**(casting operator)について説明します．RTTIを使用すると，プログラムの実行中にオブジェクトの型を判別することができます．キャスト演算子を使用すると，より安全で制御の行き届いた方法で型変換を行うことができます．キャスト演算子の1つである**dynamic_cast**はRTTIに直接関係しているので，これらの２つの主題について同じ章で説明することにします．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. 汎用関数とは何かを説明しなさい．またその一般形式を示しなさい．
2. 汎用クラスとは何かを説明しなさい．またその一般形式を示しなさい．
3. 引数の１つの指数がもう１つの引数になるまで累乗して返すgexp()という名前の汎用関数を作成しなさい．
4. 第9章9.7節の例1で，整数の座標を保持するcoordクラスを作成し，その動作を確認するプログラムを紹介しました．coordクラスを汎用クラスとして作成し，すべての型の座標を格納できるようにしなさい．また，この汎用クラスの動作を確認しなさい．
5. try，catch，throwの各文を組み合わせてC++の例外処理を行う方法について簡単に説明しなさい．
6. tryブロックを通らずにthrow文を使用することができますか．
7. terminate()関数とunexpected()関数の目的について説明しなさい．
8. すべての型の例外を処理できるcatch文の構文を示しなさい．

# 12.1　実行時型情報(RTTI)

Cのようなポリモーフィズムをサポートしない言語では実行時型情報を使用しないので，読者のみなさんにとってこれは新しい概念でしょう．ポリモーフィズムをサポートしない言語では，各オブジェク

トの型がコンパイル時(つまりプログラムの作成時)にわかっているので，実行時型情報は必要ありません．しかし，C++のようなポリモーフィズムをサポートする言語では，そのオブジェクトの正確な性質はプログラムを実行するまで決まらないので，コンパイル時にはオブジェクトの型がわからないことがあります．ご存じのとおり，C++ではクラス階層，仮想関数，基本クラスポインタを使用してポリモーフィズムを実装しています．この手法では，基本クラスポインタを使用して，基本クラスのオブジェクトや，その基本クラスから派生した任意のオブジェクトを指すことができます．したがって，どの時点で基本ポインタがどの型のオブジェクトを指すかということを前もって知ることはできません．このような情報は，実行時に実行時型情報を使用して判別しなければなりません．

オブジェクトの型を取得するには，typeid()を使用します．typeidを使用するには，<typeinfo>ヘッダを含める必要があります．typeid()の最も一般的な形式を次に示します．

**typeid()関数**

```
typeid(object)
```

*object*には，型を取得するオブジェクトを指定します．typeidはtype_info型のオブジェクトへの参照を返します．これは，object によって定義されるオブジェクトの型を記述するオブジェクトです．type_info クラスでは，次の公開メンバが定義されています．

**type_info クラス**

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name( );
```

オーバーロードされた== と!= は，型の比較を行います．before()関数は，照合順序において，仮引数に指定したオブジェクトよりも呼び出し元オブジェクトが前にある場合に，真を返します(この関数は主に内部的に使われます．この戻り値は，継承またはクラス階層とは無関係です)．name()関数は，型の名前を指すポインタを返します．

typeidを使用するとすべてのオブジェクトの型を取得することができますが，最も重要な用途は，ポリモーフィック基本クラスのポインタとして使用する方法です．この場合，typeidはポインタが指す先の実際のオブジェクトの型を自動的に返します．これは基本クラスでも，基本クラスから派生したオブジェクトでもかまいません(基本クラスポインタは，基本クラスのオブジェクトまたはその基本クラスからのすべての派生クラスを指すことができます)．したがって，typeidを使用すれば，基本クラスポインタが指すオブジェクトの型を実行時に判別することができます．同じことが参照にも当てはまります．ポリモーフィッククラスのオブジェクトへの参照に対してtypeid を適用すると，実際に参照され

ているオブジェクトの型が返されます．これは派生オブジェクト型でもかまいません．非ポリモーフィッククラスにtypeidを適用すると，ポインタまたは参照の基本型を取得できます．

typeidの形式はもう1つあります．この形式では，型名を引数として受け取ります．次にこの形式を示します．

typeid()関数

```
typeid(type-name)
```

この形式の主な用途は，指定の型について記述するtype_infoオブジェクトを取得することです．このオブジェクトを型比較文で利用することができます．

typeidは一般に間接参照ポインタ(*演算子を適用したポインタ)に適用するので，間接参照するポインタがヌルの場合に備えて，特別な例外が用意されています．このような状況では，bad_typeid例外が投げられます．

実行時型情報は，すべてのプログラムで使用するような情報ではありません．しかし，ポリモーフィック型を操作する際には，特定の状況で操作しているオブジェクトの型を調べるのに役立ちます．

## 例 12.1　実行時型情報(RTTI)

**1.** 次のプログラムは，typeidの使用例を示しています．このプログラムでは，まずC++の組み込み型であるintについての型情報を取得しています．次に，pが参照するオブジェクトの型を表示しています．pはBaseClass型のポインタです．

```
// typeidの使用例
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
  virtual void f() {}; // BaseClassクラスをポリモーフィッククラスにする
  // ...
};

class Derived1: public BaseClass {
  // ...
};

class Derived2: public BaseClass {
  // ...
};
```

```
int main()
{
  int i;
  BaseClass *p, baseob;
  Derived1 ob1;
  Derived2 ob2;

  // まず，組み込み型の型名を表示する
  cout << "Typeid of i is ";
  cout << typeid(i).name() << endl;

  // ポリモーフィック型へのtypeidの適用例
  p = &baseob;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &ob1;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &ob2;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
Typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2
```

前述のとおり，ポリモーフィック型の基本クラスポインタに対してtypeidを適用すると，ポインタが指すオブジェクトの型が実行時に判別されます．これは，プログラムの出力からもわかります．実験として，BaseClass クラスの仮想関数 f() をコメント化し，結果を観察してください．

**2.** 前述のとおり，ポリモーフィック基本クラスへの参照にtypeidを適用すると，参照先の実際のオブジェクトの型が返されます．この機能を最もよく使う状況としては，オブジェクトを参照によって関数に渡す場合があります．たとえば次のプログラムでは，WhatType() 関数では BaseClass 型のオブジェクトへの参照仮引数を宣言しています．つまり，WhatType() にはBaseClass型オブジェクトへの参照，またはBaseClass

型の派生クラスへの参照を渡すことができます．typeid演算子をこの仮引数に適用すると，渡されたオブジェクトの実際の型が返されます．

```
// 参照に対してtypeidを適用する
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
  virtual void f() {}; // BaseClassクラスをポリモーフィッククラスにする
  // ...
};

class Derived1: public BaseClass {
  // ...
};

class Derived2: public BaseClass {
  // ...
};

// 参照仮引数へのtypeidの適用例
void WhatType(BaseClass &ob)
{
  cout << "ob is referencing an object of type ";
  cout << typeid(ob).name() << endl;
}

int main()
{
  int i;
  BaseClass baseob;
  Derived1 ob1;
  Derived2 ob2;

  WhatType(baseob);
  WhatType(ob1);
  WhatType(ob2);

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
ob is referencing an object of type class BaseClass
ob is referencing an object of type class Derived1
ob is referencing an object of type class Derived2
```

**3.** オブジェクトの型名を取得する機能も状況によっては便利ですが，通常はあるオブジェクトの型がほかのオブジェクトの型と一致するかどうかを調べるだけで十分です．typeidによって返されるtype_infoオブジェクトは==演算子と!=演算子をオーバーロードするので，これは簡単に行うことができます．次のプログラムでは，これらの演算子の使用例を示しています．

```
// typeidに関する==演算子と!=演算子の使用例
#include <iostream>
#include <typeinfo>
using namespace std;
class X {
  virtual void f() {}
};

class Y {
  virtual void f() {}
};

int main()
{
  X x1, x2;
  Y y1;

  if(typeid(x1) == typeid(x2))
    cout << "x1 and x2 are same types¥n";
  else
    cout << "x1 and x2 are different types¥n";

  if(typeid(x1) != typeid(y1))
    cout << "x1 and y1 are different types¥n";
  else
    cout << "x1 and y1 are same types¥n";

  return 0;
}
```

このプログラムからの出力を次に示します．

```
x1 and x2 are same types
x1 and y1 are different types
```

**4.** これまでのサンプルプログラムではtypeidの使用例を示してきましたが，これらのプログラムで使用している型はコンパイル時にすでにわかっているので，typeidの本領が発揮されませんでした．しかし，次のプログラムでは違います．このプログラムでは，画面上に形状を描く単純なクラス階層を定義しています．階層の上部には抽象ク

ラスShapeがあります．さらに，具体的なサブクラスとしてRectangle（長方形），Triangle（三角形），Line（直線），NullShapeの4つを作成します．generator()関数は，オブジェクトを生成してそのオブジェクトを指すポインタを返します．実際にどのオブジェクトを作成するかは，乱数ジェネレータ関数rand()の結果に応じて，無作為に決定します（オブジェクトを作成する関数のことを**オブジェクトファクトリ**（object factory）と呼びます）．main()関数内では，形状を持たないNullShapeオブジェクトを除き，すべてのオブジェクトの形状を表示します．オブジェクトは無作為に作成されるため，次に作成されるオブジェクトの型を前もって知ることはできません．そこでRTTIが必要となります．

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

class Shape {
public:
  virtual void example() = 0;
};

class Rectangle: public Shape {
public:
  void example() {
    cout << "*****¥n*   *¥n*   *¥n*****¥n";
  }
};

class Triangle: public Shape {
public:
  void example() {
    cout << "*¥n* *¥n*  *¥n*****¥n";
  }
};

class Line: public Shape {
public:
  void example() {
    cout << "******¥n";
  }
};

class NullShape: public Shape {
public:
  void example() {
  }
```

```
};

// Shape派生オブジェクトのファクトリ
Shape *generator()
{
  switch(rand() % 4) {
    case 0:
      return new Line;
    case 1:
      return new Rectangle;
    case 2:
      return new Triangle;
    case 3:
      return new NullShape;
  }
  return NULL;
}

int main()
{
  int i;
  Shape *p;

  for(i=0; i<10; i++) {
    p = generator(); // 次のオブジェクトを取得する

    cout << typeid(*p).name() << endl;

    // NullShapeではない場合だけ，オブジェクトを描画する
    if(typeid(*p) != typeid(NullShape))
      p->example();
  }

  return 0;
}
```

このプログラムからの出力例を次に示します．

```
class Rectangle
*****
*   *
*   *
*****
class NullShape
class Triangle
*
* *
*  *
*****
```

```
class Line
******
class Rectangle
*****
*   *
*   *
*****
class Line
******
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Line
******
```

**5.** typeid演算子をテンプレートクラスに適用することができます．例として，次のプログラムについて考えてみましょう．このプログラムでは，値を格納するテンプレートクラスの階層を作成しています．仮想関数get_val()は，各クラスで定義されている値を返します．Numクラスの場合は数値自体の値が返されます．Squareクラスの場合は数値の2乗，Sqr_rootクラスの場合は数値の平方根が返されます．Numクラスの派生オブジェクトはgenerator()関数によって生成されます．このプログラムでは，typeid演算子を使って，生成されたオブジェクトの型を判別します．

```
// テンプレートクラスに対してtypeidを使用する例
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
  T x;
  Num(T i) { x = i; }
```

```
  virtual T get_val() { return x; };
};

template <class T>
class Square : public Num<T> {
public:
  Square(T i) : Num<T>(i) {}
  T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
  Sqr_root(T i) : Num<T>(i) {}
  T get_val() { return sqrt((double) x); }
};

// Num派生オブジェクトのファクトリ
Num<double> *generator()
{
  switch(rand() % 2 ) {
    case 0: return new Square<double> (rand() % 100);
    case 1: return new Sqr_root<double> (rand() % 100);
  }
  return NULL;
}

int main()
{
  Num<double> ob1(10), *p1;
  Square<double> ob2(100.0);
  Sqr_root<double> ob3(999.2);
  int i;

  cout << typeid(ob1).name() << endl;
  cout << typeid(ob2).name() << endl;
  cout << typeid(ob3).name() << endl;

  if(typeid(ob2) == typeid(Square<double>))
    cout << "is Square<double>¥n";

  p1 = &ob2;

  if(typeid(*p1) != typeid(ob1))
    cout << "Value is: " << p1->get_val();
  cout << "¥n¥n";

  cout << "Now, generate some Objects.¥n";
```

```
    for(i=0; i<10; i++) {
      p1 = generator(); // 次のオブジェクトを取得する
       if(typeid(*p1) == typeid(Square<double>))
         cout << "Square object: ";
       if(typeid(*p1) == typeid(Sqr_root<double>))
         cout << "Sqr_root object: ";

      cout << "Value is: " << p1->get_val();
      cout << endl;
    }

    return 0;
  }
```

このプログラムからの出力を次に示します．

```
class Num<double>
class Square<double>
class Sqr_root<double>
is Square<double>
Value is: 10000

Now, generate some Objects.
Sqr_root object: Value is: 8.18535
Square object: Value is: 0
Sqr_root object: Value is: 4.89898
Square object: Value is: 3364
Square object: Value is: 4096
Sqr_root object: Value is: 6.7082
Sqr_root object: Value is: 5.19615
Sqr_root object: Value is: 9.53939
Sqr_root object: Value is: 6.48074
Sqr_root object: Value is: 6
```

## 練習問題　12.1　実行時型情報(RTTI)

**1.** RTTI が C++ で必要とされる理由を説明しなさい．

**2.** 例 1 で述べた実験を行い，その結果を確認しなさい．

**3.** 次のプログラムコードが正しいかどうか考えなさい．

```
cout << typeid(float).name();
```

**4.** 次のプログラムコードの p が D2 オブジェクトを指しているかどうかを判別する方法を示しなさい.

```
class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
};

class D2: public B {
  void f() {}
};

int main()
{
  B *p;
```

**5.** 例 5 の Num クラスに対して，次の式が真か偽かを考えなさい.

```
typeid(Num<int>) == typeid(Num<double>)
```

**6.** 自分で RTTI の実験を行いなさい．RTTI は単純なサンプルプログラムで使うのには難解な機能ですが，非常に強力な機能であり，オブジェクトを実行時に管理するのに役立ちます.

## 12.2　dynamic_cast の使用方法

C++では，C で定義されている従来のキャスト演算子も完全にサポートされていますが，それに加えて4つのキャスト演算子が用意されています．これらはdynamic_cast，const_cast，reinterpret_cast，static_cast です．ここでは，RTTI に関係する **dynamic_cast** から説明します．残りのキャスト演算子については，以降の節で説明します．

dynamic_cast 演算子は，実行時型変換を行い，型変換の有効性を検査します．dynamic_cast の実行時に型変換が無効であれば，型変換は失敗します．次に，dynamic_cast の一般形式を示します．

**dynamic_cast 演算子**

```
dynamic_cast<target-type> (expr)
```

*target-type* には，型変換先の型を指定します．*expr* には新しい型への変換を行う式を指定します．変換先の型はポインタまたは参照型でなければならず，型変換を行う式の評価結果はポインタまたは参照にならなければなりません．したがって，dynamic_castを使って，ある型のポインタを別の型のポインタに変換したり，ある型の参照をほかの型の参照に変換したりすることができます．

dynamic_cast演算子の目的は，ポリモーフィック型の型変換を行うことです．たとえば，BとDという2つのポリモーフィック型があるとします．DクラスはBクラスから派生しています．dynamic_cast演算子を使用すれば，常にD* ポインタをB* ポインタに変換することができます．これは，基本ポインタは常に派生オブジェクトを指すことができるからです．しかし，dynamic_cast演算子を使用してB* ポインタをD* ポインタに変換することができるのは，ポインタが指すオブジェクトが実際にDオブジェクトである場合だけです．一般に，dynamic_castは型変換を行うポインタ(参照)が変換先の型のオブジェクト，または変換先の型から派生したオブジェクトである場合にだけ成功します．それ以外の場合は型変換は失敗します．ポインタに対する型変換が失敗すると，dynamic_castの評価結果はヌルになります．参照型に対するdynamic_castが失敗すると，bad_cast例外が投げられます．

次に単純な例を示します．Baseはポリモーフィッククラスであり，DerivedはBaseクラスの派生クラスであるものとします．

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // Derivedオブジェクトを指す基本ポインタ
dp = dynamic_cast<Derived *> (bp);
if(dp) cout << "Cast OK";
```

この場合，bpは実際にはDerivedオブジェクトを指しているので，基本ポインタから派生ポインタへの型変換は成功します．したがって，このプログラムコードによって「Cast OK」と表示されます．これに対して次のプログラムコードでは，bpはBaseオブジェクトを指しており，基本オブジェクトを派生オブジェクトに型変換することはできないので，型変換は失敗します．

```
bp = &b_ob; // Baseオブジェクトを指す基本ポインタ
dp = dynamic_cast<Derived *> (bp);
if(!dp) cout << "Cast Fails";
```

型変換が失敗するので，このプログラムコードの出力は「Cast Fails」となります．

状況によっては，dynamic_cast演算子をtypeidの代わりとして使用できることがあります．例として，再びBaseオブジェクトがDerivedオブジェクトのポリモーフィック基本クラスである場合を考えてみましょう．次のプログラムコードでは，bpが指すオブジェクトが実際にDerivedオブジェクトである場合にだけ，bpが指すオブジェクトのアドレスをdpに代入しています．

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp =(Derived *) bp;
```

このプログラムコードでは，実際の型変換を行うのにはC形式の型変換を使用しています．型変換を実際に行う前に，if 文を使用して型変換の有効性を調べているので，これは安全な方法です．ただし，より良い方法として，typeid 演算子と if 文を次の dynamic_cast に置き換える方法があります．

```
dp = dynamic_cast<Derived *> (bp);
```

dynamic_castは，型変換を行うオブジェクトがすでに変換先の型になっているか，その型から派生しているオブジェクトである場合にだけ成功します．したがって，この文の実行後はdpにはヌルまたはDerived型オブジェクトのポインタが格納されます．dynamic_castは型変換が有効である場合にだけ成功するので，状況によってはロジックを簡略化するのに役立ちます．

## 例 12.2 dynamic_cast の使用方法

**1.** 次のプログラムは，dynamic_cast の使用例を示しています．

```
// dynamic_castの使用例
#include <iostream>
using namespace std;

class Base {
public:
  virtual void f() { cout << "Inside Base¥n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "Inside Derived¥n"; }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Cast from Derived * to Derived * OK.¥n";
    dp->f();
```

```
} else
  cout << "Error¥n";

cout << endl;

bp = dynamic_cast<Base *> (&d_ob);
if(bp) {
  cout << "Cast from Derived * to Base * OK.¥n";
  bp->f();
} else
  cout << "Error¥n";

cout << endl;

bp = dynamic_cast<Base *> (&b_ob);
if(bp) {
  cout << "Cast from Base * to Base * OK.¥n";
  bp->f();
} else
  cout << "Error¥n";

cout << endl;

dp = dynamic_cast<Derived *> (&b_ob);
if(dp)
  cout << "Error¥n";
else
  cout << "Cast from Base * to Derived * not OK.¥n";

cout << endl;

bp = &d_ob; // bpはDerivedオブジェクトを指す
dp = dynamic_cast<Derived *> (bp);
if(dp) {
  cout << "Casting bp to a Derived * OK¥n" <<
    "because bp is really pointing¥n" <<
    "to a Derived object.¥n";
  dp->f();
} else
  cout << "Error¥n";

cout << endl;

bp = &b_ob; // bpはBaseオブジェクトを指す
dp = dynamic_cast<Derived *> (bp);
if(dp)
  cout << "Error";
else {
```

```
      cout << "Now casting bp to a Derived *¥n" <<
        "is not OK because bp is really ¥n" <<
        "pointing to a Base object.¥n";
    }

    cout << endl;

    dp = &d_ob; // dpはDerivedオブジェクトを指す
    bp = dynamic_cast<Base *> (dp);
    if(bp) {
      cout << "Casting dp to a Base * is OK.¥n";
      bp->f();
    } else
      cout << "Error¥n";

    return 0;
  }
```

このプログラムの出力は次のようになります.

```
Cast from Derived * to Derived * OK.
Inside Derived

Cast from Derived * to Base * OK.
Inside Derived

Cast from Base * to Base * OK.
Inside Base

Cast from Base * to Derived * not OK.
Casting bp to a Derived * OK
because bp is really pointing
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.

Casting dp to a Base * is OK.
Inside Derived
```

**2.** 次のプログラムは, typeidの代わりにdynamic_castを使用する方法を示しています.

```
// typeidの代わりにdynamic_castを使う
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
class Base {
public:
  virtual void f() {}
};

class Derived : public Base {
public:
  void derivedOnly() {
    cout << "Is a Derived Object¥n";
  }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  // ************************************
  // typeidを使う
  // ************************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Cast from Base to Derived failed.¥n";
  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Error, cast should work!¥n";

  // ************************************
  // dynamic_castを使う
  // ************************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Cast from Base to Derived failed.¥n";

  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
```

```
    cout << "Error, cast should work!¥n";

  return 0;
}
```

ご覧のとおり，dynamic_cast を使うと，基本ポインタを派生ポインタに型変換するためのロジックを簡略化できます．このプログラムからの出力を次に示します．

```
Cast from Base to Derived failed.
Is a Derived Object
Cast from Base to Derived failed.
Is a Derived Object
```

**3.** テンプレートクラスに対してもdynamic_cast演算子を使うことができます．たとえば，次のプログラムは前節の例5で使用したテンプレートクラスを修正したもので，generator()関数から返されたオブジェクトの型をdynamic_cast演算子を使用して判別しています．

```
// テンプレートに対してもdynamic_castを使うことができる
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
  T x;
  Num(T i) { x = i; }
  virtual T get_val() { return x; };
};

template <class T>
class Square : public Num<T> {
public:
  Square(T i) : Num<T>(i) {}
  T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
  Sqr_root(T i) : Num<T>(i) {}
  T get_val() { return sqrt((double) x); }
};

// Num派生オブジェクトのファクトリ
```

```
Num<double> *generator()
{
  switch(rand() % 2 ) {
    case 0: return new Square<double> (rand() % 100);
    case 1: return new Sqr_root<double> (rand() % 100);
  }
  return NULL;
}

int main()
{
  Num<double> ob1(10), *p1;
  Square<double> ob2(100.0), *p2;
  Sqr_root<double> ob3(999.2), *p3;
  int i;

  cout << "Generate some objects.¥n";
  for(i=0; i<10; i++) {
    p1 = generator();

    p2 = dynamic_cast<Square<double> *> (p1);
    if(p2) cout << "Square object: ";
    p3 = dynamic_cast<Sqr_root<double> *> (p1);
    if(p3) cout << "Sqr_root object: ";

    cout << "Value is: " << p1->get_val();
    cout << endl;
  }

  return 0;
}
```

## 練習問題　12.2　dynamic_cast の使用方法

**1.** dynamic_cast の目的について自分の言葉で説明しなさい.

**2.** 次のプログラムコードと dynamic_cast 演算子を使用して，ob が D2 オブジェクトである場合にだけ，p に ob オブジェクトへのポインタを代入する方法を示しなさい.

```
class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
```

```
    };

    class D2: public B {
      void f() {}
    };

    int main()
    {
      B *p;
```

**3.** 12.1 節の練習問題 4 の main() 関数を修正し，NullShape オブジェクトを表示するのを防ぐのに，typeid の代わりに dynamic_cast 演算子を使用しなさい．

**4.** この節の例 3 で示した Num クラス階層を使用した場合，次のプログラムコードが有効かどうか考えなさい．

```
Num<int> *Bp;
Square<double> *Dp;
// ...
Dp = dynamic_cast<Num<int>> (Bp);
```

## 12.3　const_cast，reinterpret_cast，static_cast の使用方法

新しいキャスト演算子の中で最も重要なのはdynamic_castですが，残りの3つの演算子も役に立ちます．これらの演算子の一般形式を次に示します．

const_cast，reinterpret_cast，static_cast 演算子

```
const_cast<target-type> (expr)
reinterpret_cast<target-ype> (expr)
static_cast<target-type> (expr)
```

*target-type*には，型変換先の型を指定します．*expr*には新しい型に変換する式を指定します．一般に，これらのキャスト演算子を使うと，C形式の型変換に比べてより安全で明示的な型変換を行うことができます．

**const_cast** 演算子を使うと，型変換における const や volatile の指定を明示的に上書きすることができます．変換先の型は，const や volatile の違いを除いて，元の型と同じでなければなりません．const_cast の最も一般的な用途は，const 指定を取り除くことです．

**static_cast**演算子は非ポリモーフィック型変換を行います．たとえば，static_cast演算子を使用して基本クラスポインタを派生クラスポインタに型変換することができます．また，static_cast 演算子はすべての標準的な変換にも使用できます．実行時の検査は行われません．

**reinterpret_cast** 演算子を使うと，あるポインタ型を基本的に異なる別のポインタ型に変更することができます．また，ポインタを整数に，整数をポインタに変換することもできます．reinterpret_cast 演算子は，本質的に互換性のないポインタ型間での型変換に使用します．

const 指定を除去できるのは const_cast 演算子だけです．つまり，dynamic_cast，static_cast，reinterpret_cast ではオブジェクトの const 指定を取り除くことはできません．

## 例 12.3　const_cast，reinterpret_cast，static_cast の使用方法

**1.** 次のプログラムは，reinterpret_cast の使用例を示しています．

```
// reinterpret_castを使用するプログラムの例
#include <iostream>
using namespace std;

int main()
{
  int i;
  char *p = "This is a string";

  i = reinterpret_cast<int> (p); // ポインタを整数に型変換する

  cout << i;
  return 0;
}
```

このプログラムでは，reinterpret_cast 演算子を使って p ポインタを整数に変換しています．これは基本的な型変換を表しており，reinterpret_cast の良い使用例です．

**2.** 次のプログラムは，const_cast の使用例を示しています．

```
// const_castを使用するプログラムの例
#include <iostream>
using namespace std;

void f(const int *p)
{
  int *v;
  // 型変換によってconst指定を取り除く
  v = const_cast<int *> (p);
```

```
  *v = 100; // vを介してオブジェクトを修正する
}

int main()
{
  int x = 99;

  cout << "x before call: " << x << endl;
  f(&x);
  cout << "x after call: " << x << endl;

  return 0;
}
```

このプログラムからの出力を次に示します．

```
x before call: 99
x after call: 100
```

ご覧のとおり，f()関数への仮引数はconstポインタとして指定されているにもかかわらず，f()関数ではxを修正しています．

const_cast演算子を使い，型変換によってconst指定を取り除くというのは，危険性のある機能です．注意して使用してください．

**3.** static_cast演算子は最初のキャスト演算子の代わりとなる演算子です．この演算子は，単に非ポリモーフィック型変換を行います．たとえば次のプログラムコードでは，floatをintに変換しています．

```
// static_castを使用するプログラムの例
#include <iostream>
using namespace std;

int main()
{
  int i;
  float f;

  f = 199.22;
  i = static_cast<int> (f);

  cout << i;

  return 0;
}
```

**練習問題 12.3　const_cast，reinterpret_cast，static_cast の使用方法**

**1.** const_cast，reinterpret_cast，static_cast を使用する目的を説明しなさい．

**2.** 次のプログラムには誤りがあります．const_cast演算子を使用してこれを修正する方法を示しなさい．

```
#include <iostream>
using namespace std;

void f(const double &i)
{
  i = 100; // エラー -- const_castを使用して修正する
}

int main()
{
  double x = 98.6;

  cout << x << endl;
  f(x);
  cout << x << endl;

  return 0;
}
```

**3.** const_cast を特別な場合にしか使用すべきでない理由を説明しなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

**1.** typeid の動作について説明しなさい．

**2.** typeid を使用するためにインクルードする必要のあるヘッダを示しなさい．

**3.** 標準的な型変換に加え，C++ では 4 つのキャスト演算子が定義されています．この 4 つの演算子を挙げ，それぞれの役割について説明しなさい．

**4.** 次に示す不完全なプログラムを完成させ，ユーザーが選択したオブジェクトの型を報告するようにしなさい．

```
#include <iostream>
#include <typeinfo>
using namespace std;

class A {
  virtual void f() {}
};

class B : public A {
};

class C: public B {
};

int main()
{
  A *p, a_ob;
  B b_ob;
  C c_ob;
  int i;

  cout << "Enter 0 for A objects, ";
  cout << "1 for B objects or ";
  cout << "2 for C objects.¥n";

  cin >> i;

  if(i==1) p = &b_ob;
  else if(i==2) p = &c_ob;
  else p = &a_ob;

  // ユーザーが選択したオブジェクトの型を報告する
  return 0;
}
```

**5.** dynamic_cast演算子をtypeidの代わりとして使用できる状況について説明しなさい.

**6.** typeid 演算子によって取得できるオブジェクトの型を示しなさい.

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう.

**1.** 12.1節の例4で示したプログラムを修正し，例外処理を使用してgenerator()関数内の割り当て失敗を監視しなさい．

**2.** 問1のgenerator()関数を修正し，new(nothrow)を使用しなさい(エラー検査を行うこと)．

**3.** 特別問題です．DataStructという名前の抽象クラスを上部に持つクラス階層を自分で作成しなさい．2つの具体的なサブクラスを作成し，1つではスタックを，もう1つではキューを実装します．次のプロトタイプを持つDataStructFactory()関数を作成しなさい．

```
DataStruct *DataStructFactory(char what);
```

DataStructFactory()関数では，whatがsの場合はスタックを作成し，whatがqの場合はキューを作成し，作成したオブジェクトを指すポインタを返します．また，このファクトリクラスの動作を確認しなさい．

# 13

# 名前空間，変換関数，その他の機能

## この章の内容

この章では，名前空間，変換関数，staticおよびconstクラスメンバ，その他のC++の特別な機能について説明します．

## 前章の理解度チェック

この章を読み進む前に、次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

1. キャスト演算子とは何か，またその目的について説明しなさい．

2. type_info とは何か説明しなさい．

3. オブジェクトの型を判別する演算子を示しなさい．

4. 次のプログラムコードがあります．p がBase オブジェクトとDerived オブジェクトのどちらを指しているかを調べる方法を示しなさい．

```
class Base {
  virtual void f() {}
};

class Derived : public Base {
};

int main()
{
  Base *p, b_ob;
  Derived d_ob;

  // ...
```

5. dynamic_cast 演算子は，型変換するオブジェクトが変換先の型，または変換先の型から＿＿＿＿＿＿＿オブジェクトのポインタである場合にだけ成功します（下線部を埋めなさい）．

6. dynamic_cast 演算子を使用して，型変換によってconst 指定を取り除くことができるかどうか述べなさい．

# 13.1　名前空間

名前空間(namespace)については第1章で簡単に説明しましたが，ここで詳しく説明することにしましょう．名前空間はC++に比較的新しく追加された機能で，その目的は識別子の名前を局所化し，名前の競合を避けることです．C++プログラミング環境では，変数，関数，クラスの名前が急増を続けてきました．名前空間が登場する前は，これらのすべての名前がグローバルな名前空間の中で場所を取り合い，多くの競合が発生していました．たとえば，プログラムでtoupper()という名前の関数を定義した場合，この関数は標準ライブラリ関数であるtoupper()と同じグローバルな名前空間に保存されるので，(仮引数リストによっては)標準ライブラリ関数が上書きされる可能性がありました．2つまたはそれ以上のサードパーティ製ライブラリを1つのプログラムで使用すると，名前の競合にさらに拍車がかかりました．このような場合，1つのライブラリで定義されている名前がほかのライブラリで定義されている同じ名前と競合する可能性が生じます(実際にこのようなことがよくありました)．

これらの問題に対する解決法として登場したのがnamespaceキーワードです．名前空間の中で定義した名前の可視性は局所化されるので，さまざまなコンテキストで同じ名前を使用しても，競合が発生することがありません．名前空間の恩恵を最も受けているのはおそらくC++標準ライブラリでしょう．初期のバージョンのC++では，C++全体がグローバルな名前空間で定義されていました(言うまでもなく，グローバルな名前空間は唯一の名前空間でした)．しかし，今ではC++ライブラリは独自の名前空間であるstd内で定義されているため，名前が競合する可能性が減りました．また，プログラム内で独自の名前空間を作成し，競合すると思われるすべての名前の可視性を局所化することができます．このことは，クラスや関数ライブラリを作成する場合には特に重要となります．

namespaceキーワードを使うと，宣言領域を作成することによって，グローバルな名前空間を区切ることができます．基本的には，名前空間はスコープ(scope，有効範囲)を定義します．namespaceキーワードの一般形式を次に示します．

**namespace キーワード**

```
namespace name {
    // 宣言
}
```

namespace 文内で定義されているものはすべて，その名前空間のスコープに含まれます．

次に，名前空間の例を示します．

```
namespace MyNameSpace {
  int i, k;
  void myfunc(int j) { cout << j; }
```

```
    class myclass {
     public:
       void seti(int x) { i = x; }
       int geti() { return i; }
    };
  }
```

この例では，変数i，変数k，myfunc()関数，およびmyclassクラスが，MyNameSpace名前空間によって定義されるスコープに含まれています．

名前空間内で定義された識別子は，その名前空間内から直接参照することができます．たとえば，MyNameSpace名前空間のreturn i文では変数iを直接使用しています．しかし，名前空間ではスコープが定義されているので，名前空間内で宣言されたオブジェクトをその名前空間の外部から参照するには，スコープ解決演算子を使用する必要があります．たとえば，MyNameSpace名前空間の外部から変数iに10という値を代入するには，次の文を使う必要があります．

```
MyNameSpace::i = 10;
```

また，MyNameSpace名前空間の外部からmyclass型のオブジェクトを宣言するには，次のような文を使用します．

```
MyNameSpace::myclass ob;
```

一般に，名前空間の外部から名前空間のメンバにアクセスするには，メンバ名の前に名前空間の名前とスコープ解決演算子を付けます．

プログラムで1つの名前空間のメンバを頻繁に参照するような場合は，そのたびに名前空間とスコープ解決演算子を指定するのは非常に面倒な作業です．この手間を軽減するために用意されたのがusing文です．using文には，次の2つの一般形式があります．

using文

```
using namespace name;
using name::member;
```

1つ目の形式の*name*には，アクセスしたい名前空間の名前を指定します．この形式を使う場合は，指定の名前空間内で定義されているすべてのメンバが現在の名前空間に取り込まれ，修飾せずに使用できるようになります．2つ目の形式を使用した場合は，その名前空間の特定のメンバだけが可視状態になります．たとえば，前述のMyNameSpace名前空間の場合は，次のusing文と代入文を使うことができます．

```
using MyNameSpace::k; // kだけが可視状態になる
k = 10; // kは可視状態なので有効
```

```
using namespace MyNameSpace; // すべてのメンバが可視状態になる
i = 10; // MyNameSpace名前空間のすべてのメンバが可視状態なので有効
```

同じ名前の名前空間宣言を複数記述することもできます．これによって，1つの名前空間を複数のファイルに分割することができ，また1つのファイル内で分割して記述することもできます．次の例について考えてみましょう．

```
namespace NS {
  int i;
}

// ...

namespace NS {
  int j;
}
```

この例では，NS名前空間を2つに分割しています．しかし，それぞれの内容は同じNSという名前空間内に存在しています．

名前空間の宣言は，ほかのすべてのスコープの外部に記述しなければなりません(ただし，1つだけ例外があり，ほかの名前空間内にネストすることはできます)．つまり，たとえば1つの関数にローカルな名前空間を宣言することはできません．

**無名名前空間**(unmamed namespace)という特殊な名前空間があります．無名名前空間を使用すると，1つのファイル内で一意の識別子を作成することができます．無名名前空間の一般形式を次に示します．

**無名名前空間**

```
namespace {
   // 宣言
}
```

無名名前空間を使用すると，1つのファイルのスコープ内でのみ認識される一意の識別子を定義することができます．つまり，無名名前空間を含むファイル内では，その名前空間のメンバを修飾せずに直接使用することができます．しかし，そのファイル外ではその識別子を認識することができません．

通常は，小，中規模のプログラムには名前空間を作成する必要はありません．しかし，再利用可能なコードのライブラリを作成する場合や，広範囲にわたる移植性を持つプログラムを作成したい場合などは，名前空間にプログラムコードを含めることも考慮するとよいでしょう．

## 例 13.1 名前空間

**1.** 次のプログラムは，名前空間の使用例を示しています．

```
// 名前空間の例
#include <iostream>
using namespace std;

// 名前空間を定義する
namespace firstNS {
  class demo {
    int i;
  public:
    demo(int x) { i = x; }
    void seti(int x) { i = x; }
    int geti() { return i; }
  };
  char str[] = "Illustrating namespaces¥n";
  int counter;
}

// ほかの名前空間を定義する
namespace secondNS {
  int x, y;
}

int main()
{
  // スコープ解決演算子を使用する
  firstNS::demo ob(10);

  /* obを宣言した後は，名前空間修飾子を使用せずに
     そのメンバ関数を使用することができる．  */
  cout << "Value of ob is : " << ob.geti();
  cout << endl;

  ob.seti(99);

  cout << "Value of ob is now : " << ob.geti();
  cout << endl;

  // strを現在のスコープに取り込む
  using firstNS::str;
  cout << str;

  // firstNSのすべてのメンバを現在のスコープに取り込む
  using namespace firstNS;
```

```
  for(counter = 10; counter; counter--)
    cout << counter << " ";
  cout << endl;

  // secondNS名前空間を使用する
  secondNS::x = 10;
  secondNS::y = 20;

  cout << "x, y: " << secondNS:: x;
  cout << ", " << secondNS::y << endl;

  // 2つ目の名前空間を可視状態にする
  using namespace secondNS;
  demo xob(x), yob(y);

  cout << "xob, yob: " << xob.geti() << ", ";
  cout << yob.geti() << endl;

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
Value of ob is : 10
Value of ob is now : 99
Illustrating namespaces
10 9 8 7 6 5 4 3 2 1
x, y: 10, 20
xob, yob: 10, 20
```

このプログラムから重要なことが1つわかります．それは，ある名前空間を使用しても，ほかの名前空間が上書きされることはないということです．名前空間を可視状態にすると，その名前空間のメンバが現在有効なほかの名前空間に追加されるだけのことです．したがって，このプログラムの終わりまでには，std，firstNS，secondNSの各名前空間がグローバルな名前空間に追加されたことになります．

**2.** 前述のとおり，1つの名前空間を複数のファイルに分割したり，1つのファイル内で分割して記述したりすることができます．それぞれの内容は累積されます．次に例を示します．

```
// 名前空間は累積される
#include <iostream>
using namespace std;

namespace Demo {
  int a; // Demo名前空間内
```

```
}

int x;    // グローバルな名前空間内

namespace Demo {
  int b; // これもDemo名前空間内
}

int main()
{
  using namespace Demo;

  a = b = x = 100;

  cout << a << " " << b << " " << x;

  return 0;
}
```

この例では，aとbはDemo名前空間に含まれていますが，xは含まれていません．

**3.** 前述のとおり，標準C++のライブラリ全体はstdという専用の名前空間で定義されています．本書で紹介する大部分のプログラムに，次の文が含まれているのはこのためです．

```
using namespace std;
```

この文によって，std名前空間が現在の名前空間に取り込まれ，ライブラリ内で定義されている関数およびクラスの名前にいちいちstd::を付けることなく，直接アクセスできるようになります．

当然ながら，それぞれの名前にstd::を付けて明示的に修飾することもできます．たとえば，次のプログラムでは，ライブラリをグローバルな名前空間に取り込んでいません．

```
// 明示的なstd::修飾子を使用する
#include <iostream>

int main()
{
  double val;

  std::cout << "Enter a number: ";
  std::cin >> val;
  std::cout << "This is your number: ";
  std::cout << val;
```

```
  return 0;
}
```

この例では，coutとcinを明示的に修飾し，名前空間を指定しています．つまり，標準出力に書き込むにはstd::coutと記述し，標準入力から読み込むにはstd::cinと記述する必要があります．

標準C++ ライブラリをそれほど使用しないプログラムでは，標準C++ ライブラリをグローバルな名前空間に取り込む必要はありません．しかし，ライブラリのメンバを数百回も使用する場合には，stdを現在の名前空間に含める方が，それぞれの名前を個々に修飾するよりもはるかに簡単です．

**4.** 標準ライブラリのメンバを少数しか使用しない場合は，各メンバごとにusing文を指定する方が効率的です．この方法の利点は，標準ライブラリ全体をグローバルな名前空間に取り込むことなく，std::修飾子を使わずにそれらの名前を使用できるということです．次に例を示します．

```
// 少しの名前だけをグローバルな名前空間に取り込む
#include <iostream>

// coutとcinへのアクセスを可能にする
using std::cout;
using std::cin;

int main()
{
  double val;
  cout << "Enter a number: ";

  cin >> val;
  cout << "This is your number: ";
  cout << val;

  return 0;
}
```

このプログラムではcinとcoutを直接使用することができますが，std名前空間のほかの名前は直接使用することができません．

**5.** 前述のとおり，最初のC++ ライブラリはグローバルな名前空間で定義されていました．古いC++プログラムを変換する際には，using namespace std文を含めるか，それぞれのライブラリメンバ参照をstd::で修飾する必要があります．古い.hヘッダファイルを新しいスタイルのヘッダに置き換える場合には，このことが特に重要です．古

い.hヘッダの内容はグローバルな名前空間に追加されることに注意してください．新しいスタイルのヘッダの場合は，その内容はstd名前空間に追加されます．

**6.** グローバル名のスコープを宣言ファイル内だけに限定したい場合，Cではその名前をstaticとして宣言していました．例として，次の2つのファイルを考えてみましょう．これらは同じプログラムの一部です．

**ファイル1**

```
static int counter;
void f1() {
  counter = 99; // 有効
}
```

**ファイル2**

```
extern int counter;
void f2() {
  counter = 10; // エラー
}
```

counterはファイル1で定義されているので，ファイル1で使用することができます．ファイル2では，counterはexternとして指定されていますが，まだ利用することができず，利用しようとするとエラーが発生します．ファイル1のcounterの前にstaticを付けることにより，counterのスコープをこのファイル内だけに限定することができます．

staticグローバル宣言の使用はC++でも許可されていますが，無名名前空間を使用する方法の方が優れており，同じ結果を得ることができます．次に例を示します．

**ファイル1**

```
namespace {
  int counter;
}
void f1() {
  counter = 99; // OK
}
```

**ファイル2**

```
extern int counter;
void f2() {
  counter = 10; // error
}
```

この例では，counter はファイル1内に限定されています．標準C++では，static の代わりに無名名前空間を使う方法が推奨されています．

## 練習問題 13.1 名前空間

**1.** 第9章で使用した次のプログラムを，using namespace std 文を使用しないように修正しなさい．

```
// 空白を|に変換する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CONVERT <input> <output>¥n";
    return 1;
  }
  ifstream fin(argv[1]); // 入力ファイルを開く
  ofstream fout(argv[2]);  // 出力ファイルを作成する

  if(!fout) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  if(!fin) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char ch;

  fin.unsetf(ios::skipws);  // 空白を飛ばさない
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '|';
    if(!fin.eof()) fout << ch;
  }

  fin.close();
  fout.close();
```

```
        return 0;
    }
```

2. 無名名前空間の動作を説明しなさい.

3. usingの2つの形式の違いを説明しなさい.

4. 本書で紹介する大部分のプログラムにusing文が含まれている理由を説明しなさい. また，代替法を1つ説明しなさい.

5. 再利用可能なプログラムコードを作成する場合に，専用の名前空間に含めた方がよい理由を説明しなさい.

## 13.2　変換関数の作成方法

ある型のオブジェクトをほかの型のオブジェクトに変換すると便利なことがあります. オーバーロード演算子関数を使用してそのような変換を行うことは可能ですが，さらに簡単な(そして優れた)方法もあります. それは変換関数を使用する方法です. **変換関数**(conversion function)は，オブジェクトをほかの型(通常はC++の組み込み型のいずれか)と互換性のある値に変換します. 基本的に変換関数は，オブジェクトを使用する式の型と互換性のある値に，そのオブジェクトを自動的に変換します.

変換関数の一般形式を次に示します.

**変換関数**

```
operator type( ) { return value; }
```

*type*には変換先の型を指定し，*value*には変換を実行した後のオブジェクトの値を指定します. 変換関数はtype型の値を返します. 仮引数を指定することはできず，変換関数は変換を行うクラスのメンバでなければなりません.

これから示す例を見てもわかるとおり，C++で利用できるほかの方法と比べて，変換関数を使うと最も明確な方法でオブジェクトの値をほかの型に変換することができます. これは，変換関数を使用すると，変換先の型を使用する式の中にオブジェクトを直接含めることができるからです.

## 例 13.2　変換関数の作成方法

**1.** 次のプログラムのcoordクラスには，オブジェクトを整数に変換する変換関数が含まれています．この例の関数は2つの座標の積を返しますが，作成するアプリケーションに応じて適切な変換を行うことができます．

```
// 単純な変換関数の例
#include <iostream>
using namespace std;

class coord {
  int x, y;
public:
  coord(int i, int j) { x = i; y = j; }
  operator int() { return x*y; } // 変換関数
};

int main()
{
  coord o1(2, 3), o2(4, 3);
  int i;

  i = o1;  // 自動的に整数に変換する
  cout << i << '¥n';

  i = 100 + o2;  // o2を整数に変換する
  cout << i << '¥n';

  return 0;
}
```

このプログラムによって，6 と 112 が表示されます．

この例では，o1 を整数に代入したときと，o2 を大きな整数式の一部として使用したときに変換関数が呼び出されています．前述のとおり，変換関数を使うと，一連の複雑なオーバーロード演算子関数を作成することなく，自分で作成したクラスを「通常の」C++ 式に統合することができます．

**2.** 次に，変換関数の例をもう1つ示します．この例では，strtype 型の文字列を str を指す文字ポインタに変換しています．

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len = strlen(s); }
  operator char *() { return str; } // char*に変換する
};

int main()
{
  strtype s("This is a test¥n");
  char *p, s2[80];

  p = s; // char *に変換する
  cout << "Here is string: " << p << '¥n';

  // 関数呼び出しの中でchar *に変換する
  strcpy(s2, s);
  cout << "Here is copy of string: " << s2 << '¥n';

  return 0;
}
```

このプログラムからの出力は，次のようになります．

```
Here is string: This is a test
Here is copy of string: This is a test
```

ご覧のとおり，変換関数はオブジェクトsをp（char * 型）に代入する際だけでなく，strcpy()関数への仮引数としてsを使用するときにも呼び出されています．strcpy()関数のプロトタイプは，次のとおりです．

**strcpy()関数**

```
char *strcpy(char *s1, const char *s2);
```

プロトタイプでは，*s2*がchar * 型であることを指定しているため，char * への変換関数が自動的に呼び出されます．このことから，変換関数を使うと，独自のクラスをC++の標準ライブラリ関数にスムーズに統合できることがおわかりいただけたでしょう．

**練習問題 13.2 変換関数の作成方法**

**1.** 例2で紹介したstrtypeクラスを使用して，int型への変換を行う変換関数を作成しなさい．この関数では，strに保存している文字列の長さを返します．また，この変換関数の動作を確認しなさい．

**2.** 次のクラスで，pwr型オブジェクトを整数型に変換する変換関数を作成しなさい．関数からは$base^{exp}$の結果を返します．

```
class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
  // ここに整数への変換関数を作成する
};
```

## 13.3　static クラスメンバ

クラスメンバ変数をstaticとして宣言することができます．staticメンバ変数を使用すると，数多くの厄介な問題を回避することができます．メンバ変数をstaticとして宣言すると，そのクラスのオブジェクトをいくつ作成したとしても，その変数のコピーは1つしか存在しなくなります．各オブジェクトは，単に1つの変数を共有します．通常のメンバ変数の場合は，オブジェクトを作成するたびに変数が新しく作成され，このコピーにはそのオブジェクトからしかアクセスすることができません(つまり，通常の変数を使用するときは，各オブジェクトが専用のコピーを持ちます)．これに対してstaticメンバ変数のコピーは1つしか存在せず，そのクラスのすべてのオブジェクトがこの変数を共有します．また，1つのstatic変数を，そのstaticメンバを含むクラスから派生したすべてのクラスで使用することができます．

最初は奇妙に思えるかもしれませんが，staticメンバ変数はそのクラスのどのオブジェクトを作成するよりも前から存在します．基本的に，staticメンバは，宣言されたクラス内にスコープが限定されているというだけのグローバル変数です．この後で例を紹介しますが，実際にどのオブジェクトにも依存せずにstaticメンバ変数にアクセスすることが可能です．

クラス内でstaticデータメンバを宣言しても，まだそのメンバを定義したわけではありません．staticデータメンバの定義はクラス外の別の場所で記述しなければなりません．このためには，スコープ解決演算子を使って所属するクラスを示し，static変数を再宣言します．

すべてのstaticメンバ変数は，デフォルトでは0に初期化されます．ただし，必要であれば，staticクラス変数に任意の初期値を設定することも可能です．

C++でstaticメンバ変数がサポートされている主な理由は，グローバル変数の必要性をなくすことです．グローバル変数に依存するクラスは，OOPとC++の基本であるカプセル化の原則に反します．

メンバ関数をstaticとして宣言することもできますが，あまり一般的ではありません．staticとして宣言されたメンバ関数は，そのクラスのほかのstaticメンバにしかアクセスできません(当然ながら，staticメンバ関数は非staticグローバルデータと関数にアクセスできます)．staticメンバ関数にはthisポインタがありません．仮想staticメンバ関数は作成できません．また，staticメンバ関数をconstまたはvolatileとして宣言することはできません．staticメンバ関数は，そのクラスのオブジェクトから呼び出すことができます．また，クラス名とスコープ解決演算子を使用すれば，どのオブジェクトにも依存せずにstaticメンバ関数を呼び出すことができます．

## 例 13.3 staticクラスメンバ

**1.** 次のプログラムは，staticメンバ変数を使用する単純な例です．

```
// staticメンバ変数の例
#include <iostream>
using namespace std;

class myclass {
  static int i;
public:
  void seti(int n) { i = n; }
  int geti() { return i; }
};

// myclass::iの定義．iはまだmyclassクラスに対して非公開
int myclass::i;

int main()
{
  myclass o1, o2;

  o1.seti(10);
  cout << "o1.i: " << o1.geti() << '¥n'; // 10を表示する
  cout << "o2.i: " << o2.geti() << '¥n'; // これも10を表示する

  return 0;
}
```

このプログラムからの出力は次のようになります．

```
o1.i: 10
o2.i: 10
```

このプログラムでは，static メンバ i の値を実際に設定しているのはオブジェクト o1 だけです．しかし，i は o1 と o2 の両方で(実際には myclass 型のすべてのオブジェクト間で)共有されているので，どちらの geti() 関数を呼び出しても同じ結果が表示されます．

myclass クラス内で i を宣言し，このクラス外で定義している点に注目してください．この 2 つの手順によって，変数 i の記憶域が定義されます．技術的にいうと，クラス宣言は単なる宣言にすぎません．宣言だけではメモリは確保されません．static データメンバにはメモリが割り当てられているものなので，記憶域を割り当てるための定義が別に必要となります．

**2.** static メンバ変数は，そのクラスのオブジェクトを作成するよりも前から存在するので，どのオブジェクトにも依存することなくプログラム内からアクセスすることができます．たとえば，次に示すのは例1のプログラムを修正したもので，特定のオブジェクトを参照することなく変数 i の値を 10 に設定しています．変数 i にアクセスするには，スコープ解決演算子とクラス名を使用しています．

```
// どのオブジェクトにも依存せずにstaticメンバ変数を使用する
#include <iostream>
using namespace std;

class myclass {
public:
  static int i;
  void seti(int n) { i = n; }
  int geti() { return i; }
};

int myclass::i;

int main()
{
  myclass o1, o2;

  // iを直接設定する
  myclass::i = 100; // オブジェクトを参照していない

  cout << "o1.i: " << o1.geti() << '¥n'; // 100を表示する
  cout << "o2.i: " << o2.geti() << '¥n'; // これも100を表示する

  return 0;
}
```

i は100に設定されているので，次の出力が生成されます．

```
o1.i: 100
o2.i: 100
```

**3.** static クラス変数のごく一般的な用途として，ディスクファイルやプリンタ，ネットワークサーバーなどの共有リソースへのアクセスを調整することが挙げられます．これまでのプログラミング経験からご存じかもしれませんが，1つの共有リソースへのアクセスを調整するには，イベントを順次処理するための何らかの手段が必要となります．static メンバ変数を使用して共有リソースへのアクセスを制御するという概念を理解するために，次のプログラムを見てみましょう．このプログラムではoutputというクラスを作成しています．このクラスでは，outbuf という名前の共通出力バッファを管理します．これ自身がstatic文字配列です．このバッファでは，outbuf() メンバ関数が送信した出力を受け取ります．outbuf() メンバ関数は，str の内容を一度に1文字ずつ送信します．そのためには，まずバッファへのアクセスを取得し，それから str 内のすべての文字を送信します．出力が終了するまで，ほかのオブジェクトからバッファへのアクセスはロックされます．次のプログラムコードとコメントを読めば，この動作の流れを理解することができるでしょう．

```
// リソース共有の例
#include <iostream>
#include <cstring>
using namespace std;

class output {
  static char outbuf[255]; // これは共有リソース
  static int inuse;        // 0の場合はバッファを利用できる
  static int oindex;       // outbufの索引
  char str[80];
  int i;   // str内の次のcharの索引
  int who; // オブジェクトを識別する．0より大きくなければならない
public:
  output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

  /* この関数は，バッファを待機している場合は-1を返す
     出力を完了した場合は0,
     バッファを使用中の場合はwhoを返す
  */
  int putbuf()
  {
    if(!str[i]) { // 出力完了
      inuse = 0;  // バッファを解放する
      return 0;   // 終了を通知する
```

```
      }
      if(!inuse) inuse = who;      // バッファを取得する
      if(inuse != who) return -1; // ほかのオブジェクトが出力中
      if(str[i]) { // まだ出力する文字がある
        outbuf[oindex] = str[i];
        i++; oindex++;
        outbuf[oindex] = '¥0';  // 常にヌルで終了する
        return 1;
      }
      return 0;
    }
    void show() { cout << outbuf << '¥n'; }
};

char output::outbuf[255]; // これは共有リソース
int output::inuse = 0;    // 0の場合はバッファを利用できる
int output::oindex = 0;   // outbufの索引

int main()
{
  output o1(1, "This is a test"), o2(2, " of statics");
  while(o1.putbuf() | o2.putbuf()) ; // 文字を出力する
  o1.show();

  return 0;
}
```

**4.** static メンバ関数の用途は限られていますが，優れた用途として，実際にオブジェクトを作成する前に非公開staticデータを「事前初期化」しておくことが挙げられます．たとえば，次に示すC++ プログラムは完全に有効です．

```
#include <iostream>
using namespace std;

class static_func_demo {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_func_demo::i; // iを定義する

int main()
{
  // オブジェクトを作成する前にstaticデータを初期化する
  static_func_demo::init(100);
  static_func_demo x;
```

```
    x.show(); // 100を表示する

    return 0;
  }
```

このプログラムでは，static_func_demoを作成する前に，init()関数を呼び出して変数iを初期化しています．

**練習問題 13.3 staticクラスメンバ**

1. 例3のプログラムを修正し，文字を出力中のオブジェクトと，バッファが使われているために文字の出力をブロックされているオブジェクトを表示しなさい．

2. staticメンバ変数の興味深い用途の1つとして，特定の時点で存在する特定のクラスのオブジェクトの数を追跡するというものがあります．このためには，クラスのコンストラクタが呼び出されるたびにstaticメンバをインクリメントし，クラスのデストラクタが呼び出されるたびにそれをデクリメントするという方法をとります．このようなプログラムを作成し，その動作を確認しなさい．

## 13.4　constメンバ関数とmutable

クラスメンバ関数は，constとして宣言することができます．**const**として宣言すると，その関数では呼び出し元のオブジェクトを修正できなくなります．また，constオブジェクトからは非constメンバ関数を呼び出すことができません．しかし，constメンバ関数はconstオブジェクトからも非constオブジェクトからも呼び出すことができます．

メンバ関数をconstとして指定するには，次の例に示す形式を使用します．

```
class X {
  int some_var;
public:
  int f1() const; // constメンバ関数
};
```

ご覧のとおり，関数の仮引数宣言の後にconstが続いています．

場合によっては，const関数からクラスの1つまたは複数のメンバを修正できるようにし，それ以外のメンバは修正できないようにしたいことがあります．このような場合は**mutable**を使用します．これによってconst指定が上書きされます．つまり，mutableメンバはconstメンバ関数から修正することができます．

## 例 13.4　const メンバ関数と mutable

**1.** メンバ関数をconstとして宣言する目的は，メンバ関数を呼び出したオブジェクトを修正できないようにすることです．次に例を示します．

```
/*
   constメンバ関数の使用例
   このプログラムはコンパイルできない
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // 有効
  }

  void seti(int x) const {
    i = x; // エラー
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

seti()関数はconstとして宣言されているので，このプログラムをコンパイルすることはできません．つまり，呼び出し元オブジェクトを修正することが許可されていません．この関数ではiを修正しようとしているので，プログラムはエラーとなります．これに対して，geti()関数ではiを修正しようとしていないので，この関数は完全に有効です．

**2.** 選択されたメンバをconstメンバ関数から修正できるようにするには，mutableを使用してメンバを宣言します．次に例を示します．

```
// mutableの使用例
#include <iostream>
```

```
using namespace std;

class Demo {
  mutable int i;
  int j;
public:
  int geti() const {
    return i; // 有効
  }

  void seti(int x) const {
    i = x; // 今度は有効
  }

/* 次の関数はコンパイルできない
  void setj(int x) const {
    j = x; // まだエラー
  }
*/
};

int main()
{
  Demo ob;
  ob.seti(1900);
  cout << ob.geti();
  return 0;
}
```

このプログラムでは変数iをmutableとして宣言しているので，seti()関数から変更することができます．しかしjはmutableではないので，setj()関数からjの値を修正することはできません．

## 練習問題　13.4　constメンバ関数とmutable

**1.** 次のプログラムでは，特定の時間が経過したときに音を鳴らす単純なカウントダウンタイマーを作成しています．時間とインクリメント値は，CountDownオブジェクトの作成時に指定することができます．残念ながら，このプログラムはこのままではコンパイルできません．このプログラムを修正しなさい．

```
// このプログラムにはエラーがある
#include <iostream>
using namespace std;
```

```
class CountDown {
  int incr;
  int target;
  int current;
public:
  CountDown(int delay, int i=1) {
    target = delay;
    incr = i;
    current = 0;
  }

  bool counting() const {
    current += incr;

    if(current >= target) {
      cout << "¥a";
      return false;
    }
    cout << current << " ";

    return true;
  }
};

int main()
{
  CountDown ob(100, 2);
  while(ob.counting()) ;
  return 0;
}
```

**2.** const メンバ関数から非 const 関数を呼び出すことができるかどうか述べなさい．できない場合は，なぜできないか説明しなさい．

# 13.5　コンストラクタについてのその他の事項

コンストラクタについては本書の最初の方でも説明しましたが，さらに説明しておかなければならない点があります．次のプログラムについて考えてみましょう．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
```

```
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
  myclass ob(4);

  cout << ob.geta();

  return 0;
}
```

このプログラムのmyclassコンストラクタは仮引数を1つ受け取ります．main()関数内でのobの宣言に注目してください．obの後に括弧で囲んで指定している4はmyclass()関数のx仮引数に渡される引数で，これを使用してaを初期化します．これは，本書で最初から使用してきた初期化の形態です．しかし，このほかにも方法があります．たとえば，次の文でもaを4に初期化することができます．

```
myclass ob = 4; // myclass(4)に自動的に変換する
```

コメントに示したとおり，この形態の初期化は，4を引数としたmyclassコンストラクタへの呼び出しに自動的に変換されます．つまり，コンパイラによって，この文は次のように書いた場合と同じように処理されます．

```
myclass ob(4);
```

一般には，引数を1つしか必要としないコンストラクタがある場合は，常にob(x)またはob = xを使用してオブジェクトを初期化することができます．その理由は，1つの引数を受け取るコンストラクタを作成した場合は，引数の型をそのクラスの型に変換する変換関数を暗黙的に作成していることになるからです．

暗黙的な変換関数を作成したくない場合は，explicitを使用することによってそれを防ぐことができます．explicit指定子はコンストラクタに対してしか使用できません．explicitを指定されたコンストラクタは，通常のコンストラクタ構文によって初期化を行う場合にしか使用しません．自動変換は行われません．たとえば，myclassコンストラクタがexplicitとして宣言されている場合，自動変換は行われません．myclass()関数をexplicitとして宣言する方法を次に示します．

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
```

```
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

この場合は，次の形式のコンストラクタしか使用できません．

```
myclass ob(110);
```

## 例 13.5 コンストラクタについてのその他の事項

**1.** 1つのクラスに，複数の変換コンストラクタがある場合があります．たとえば，次のmyclassクラスを考えてみましょう．

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  myclass(char *str) { a = atoi(str); }
  int geta() { return a; }
};

int main()
{
  myclass ob1 = 4;      // myclass(4)に変換する
  myclass ob2 = "123"; // myclass("123")に変換する

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;

  return 0;
}
```

どちらのコンストラクタも異なる型の引数を使っているので(当然ながら必須要件です)，それぞれの初期化文は対応するコンストラクタ呼び出しに自動的に変換されます．

**2.** コンストラクタの最初の引数の型からコンストラクタ自体の呼び出しへの自動変換では，興味深い事柄が示唆されています．たとえば，例1のmyclassクラスに対して次のmain()関数を使う場合を考えてみましょう．このmain()関数では，intおよびchar *からの変換を行って，ob1とob2に新しい値を代入しています．

```
int main()
{
  myclass ob1 = 4;      // myclass(4)に変換する
  myclass ob2 = "123"; // myclass("123")に変換する

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;

  // 自動変換を使用して新しい値を代入する
  ob1 = "1776"; // ob1 = myclass("1776")に変換する
  ob2 = 2001;   // ob2 = myclass(2001)に変換する

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;

  return 0;
}
```

**3.** コンストラクタにexplicitを指定すると，上記のような変換を防ぐことができます．次に例を示します．

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  explicit myclass(char *str) { a = atoi(str); }
  int geta() { return a; }
};
```

## 練習問題　13.5　コンストラクタについてのその他の事項

**1.** 例3で，myclass(int)だけをexplicitとした場合，myclass(char *)によって暗黙的な変換が行われるかどうか考えなさい(実際に試しなさい)．

**2.** 次のプログラムコードが有効かどうか考えなさい．

```
class Demo {
  double x
public:
  Demo(double i) { x = i; }
  // ...
```

```
    };

    // ...
    Demo counter = 10;
```

**3.** explicitキーワードの意義(C++で，状況によっては暗黙的なコンストラクタ変換が望ましくない理由)について説明しなさい．

## 13.6　リンケージ指定子とasmキーワードの使用方法

C++には，C++とほかの言語とのリンクを容易にするために，2つの重要な機構が用意されています．その1つは**リンケージ指定子**(linkage specifier)で，これによって，C++プログラム内の1つまたは複数の関数を他言語(命名や仮引数引き渡し，スタック回復などを異なる手法で実行する)にリンクするように，コンパイラに伝えることができます．もう1つは**asm**キーワードで，これによってアセンブリ言語の命令をC++ソースコードに埋め込むことができます．ここではこの2つの機能について説明します．

デフォルトでは，C++のすべての関数はC++関数としてコンパイルされ，リンクされます．しかし，ほかの種類の言語と互換性を持つ関数としてリンクするように，コンパイラに指示することもできます．すべてのC++コンパイラでは，関数をC関数としてもC++関数としてもリンクすることができます．また，関数をPascalやAda，FORTRANなどの言語とリンクできるコンパイラもあります．関数を他言語とリンクするにはリンケージ指定子を使用します．リンケージ指定子の一般形式を次に示します．

**リンケージ指定子**

```
extern "language" function-prototype;
```

*language*には，指定の関数をリンクする言語の名前を指定します．複数の関数のリンケージを指定したい場合は，次の形式のリンケージ指定子を使用します．

**リンケージ指定子**

```
extern "language" {
   function-prototypes
}
```

すべてのリンケージ指定はグローバルでなければなりません．1つの関数内でリンケージ指定を使用することはできません．

リンケージ指定子を最もよく使うのは，C++ プログラムをCのプログラムコードにリンクするときです．「C」リンケージを指定することによって，コンパイラが型情報の埋め込まれた関数名の**マングリング**(mangling)（**装飾**(decorating) とも呼ばれる)を行うのを防ぐことができます．C++ では関数をオーバーロードしたりメンバ関数を作成したりすることができるので，関数のリンク名には型情報を追加するのが一般的です．オーバーロードやメンバ関数がサポートされていないCでは，マングル名(装飾名)を認識することができません．「C」リンケージを使用すれば，この問題を回避できます．

アセンブリ言語のルーチンとC++ プログラムをリンクすることも一般には可能ですが，アセンブリ言語を使う場合にはさらに簡単な方法があります．C++ にはasm という特殊なキーワードがあり，これを使用するとアセンブリ言語の命令をC++ 関数内に含めることができます．これらの命令はそのままコンパイルされます．インラインアセンブラを使う利点は，プログラム全体を完全にC++ として定義することができ，アセンブリ言語ファイルを別個にリンクする必要がないということです．次に，asm キーワードの一般形式を示します．

asm キーワード

```
asm ("op-code");
```

*op-code* には，プログラム内に埋め込むアセンブリ言語命令を指定します．

また，一部のコンパイラでは，次の3種類の形式のasm 文もサポートされています．

asm 文

```
asm op-code;
asm op-code newline
asm {
   instruction sequence
}
```

これらの形式では，op-codeを二重引用符で囲みません．埋め込まれたアセンブリ言語命令はコンパイラの実装方法に依存するので，詳細についてはコンパイラのユーザーマニュアルを参照してください．

メモ　Microsoft Visual C++ では，アセンブリコードを埋め込むために_ _asm を使用します．これ以外の点はasm とよく似ています．

## 例 13.6 リンケージ指定子と asm キーワードの使用方法

**1.** 次のプログラムでは，func()関数をC++関数ではなくC関数としてリンクしています.

```
// リンケージ指定子の使用例
#include <iostream>
using namespace std;

extern "C" int func(int x); // C関数としてリンクする

// この関数はC関数としてリンクされる
int func(int x)
{
  return x/3;
}
```

この関数は，Cコンパイラによってコンパイルされたプログラムコードとリンクすることができます.

**2.** 次のプログラムコードでは，f1()，f2()，f3()をC関数としてリンクすることをコンパイラに通知しています.

```
extern "C" {
  void f1();
  int f2(int x);
  double f3(double x, int *p);
}
```

**3.** 次のプログラムコードでは，いくつかのアセンブリ言語命令をfunc()関数に埋め込んでいます.

```
// この関数を実行しないこと！
void func()
{
  asm("mov bp, sp");
  asm("push ax");
  asm("mov cl, 4");
  // ...
}
```

**重要** インラインアセンブリ言語をうまく使うには，アセンブリ言語についての豊富な知識が必要です．また，コンパイラのユーザーズマニュアルで，アセンブリ言語の使用法についての詳細を調べてください.

**練習問題 13.6 リンケージ指定子とasmキーワードの使用方法**

1. コンパイラのユーザーズマニュアルで，リンケージ指定とアセンブリ言語インターフェイスについて調べ，自分で学習しなさい．

# 13.7　配列ベースの入出力

コンソールおよびファイル入出力に加えて，C++では文字配列を入力/出力デバイスとして使用する一連の関数がサポートされています．C++の配列ベースの入出力(array-based I/O)は，概念的にはCの配列ベース入出力と似ていますが(特に，Cのsscanf()関数とsprintf()関数)，C++の配列ベースの入出力にはユーザー定義型を統合することができるので，はるかに柔軟で使いやすくなっています．ここで配列ベースの入出力についてすべてを説明することはできませんが，最も重要でよく使われる機能について説明することにします．

最初に，配列ベースの入出力もやはりストリームを介して動作することを理解しておいてください．第8章と第9章でC++の入出力について学んだ内容は，すべて配列ベースの入出力にも当てはまります．実際に，ほんの少しの新しい関数を学べば，配列ベースの入出力を最大限に利用することができます．これらの関数は，ストリームをメモリのある領域にリンクします．それ以降，すべての入出力はすでに学んだ入出力関数を通して発生します．

配列ベースの入出力を使うには，<strstream>ヘッダをファイルにインクルードしなければなりません．このヘッダでは，istrstream，ostrstream，strstreamの各クラスが定義されています．これらのクラスはそれぞれ，入力，出力，入出力ストリームを作成します．これらのクラスはiosを基本クラスとしているので，istream，ostream，iostreamに含まれるすべての関数およびマニピュレータはistrstream，ostrstream，strstreamでも使用できます．

出力用の文字配列を使用するには，次のようなostrstreamコンストラクタの一般形式を使用します．

**ostrstream コンストラクタ**

```
ostrstream ostr (char *buf, streamsize size, openmode mode =
➡ios::out);
```

ostrには，配列bufに関連付けるストリームを指定します．sizeには配列のサイズを指定します．modeにはデフォルトのoutputを使うのが一般的ですが，必要であれば，iosクラスで定義されている任意の出力モードフラグを指定することもできます(詳細については第9章を参照)．

出力用の配列を開いたら，配列がいっぱいになるまで文字が収められます．配列のサイズを超えて文字を収めることはできません．配列のサイズを超えて文字を格納しようとすると，入出力エラーが発生します．配列に書き込まれた文字数を調べるには，次に示すpcount()メンバ関数を使用します．

**pcount()メンバ関数**

```
streamsize pcount( );
```

この関数はストリームと組み合わせて呼び出す必要があり，配列に書き込まれた文字数(ヌル終端文字を含む)を返します．

入力用の配列を開くには，次の形式のistrstreamコンストラクタを使用します．

**istrstream コンストラクタ**

```
istrstream istr (const char *buf );
```

bufには，入力に使用する配列を指すポインタを指定します．入力ストリームはistrという名前になります．配列から入力を読み取るとき，配列の最後に達するとeof()関数は真を返します．

入出力操作用の配列を開くには，次の形式のstrstreamコンストラクタを使用します．

**strstream コンストラクタ**

```
strstream iostr (char *buf, streamsize size, openmode mode =
➡ios::in | ios::out);
```

*iostr*は，bufが指す配列を使用する入出力ストリームです．bufは，sizeに指定した文字長の配列です．

C++に関する書籍では，文字ベースのストリームを**char * ストリーム**(char * stream) と呼ぶこともあります．

バイナリ入出力関数やランダムアクセス関数など，前に説明したすべての入出力関数は，配列ベースの入出力でも動作することを覚えておいてください．

メモ

文字ベースのストリームクラスの使用は，標準C++では推奨されていません．つまり，まだ有効ではありますが，将来のC++言語ではサポートされない可能性があります．今のところはまだ広く使われているので本書でも紹介しましたが，新しいプログラムコードを作成する際には，第14章で説明するコンテナのいずれかを使用した方がよいでしょう．

## 例 13.7　配列ベースの入出力

1. 次のプログラムでは，出力用の配列を開き，その中にデータを書き込んでいます．

```
// 配列ベースの出力を使用した短いサンプルプログラム
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[255]; // 出力用バッファ

  ostrstream ostr(buf, sizeof buf); // 出力配列を開く

  ostr << "Array-based I/O uses streams just like ";
  ostr << "'normal' I/O¥n" << 100;
  ostr << ' ' << 123.23 << '¥n';

  // マニピュレータを使うこともできる
  ostr << hex << 100 << ' ';

  // 書式フラグも使用できる
  ostr.setf(ios::scientific);
  ostr << dec << 123.23;
  ostr << endl << ends;

  // 結果の文字列を表示する
  cout << buf;

  return 0;
}
```

このプログラムからの出力は，次のようになります．

```
Array-based I/O uses streams just like 'normal' I/O
100 123.23
64 01.2323e+02
```

ご覧のとおり，配列ベースの入出力を使用する場合でも，オーバーロード入出力演算子，組み込み入出力マニピュレータ，メンバ関数，書式フラグをすべて問題なく使用できます(独自のクラスに対して相対的に作成したマニピュレータやオーバーロード入出力演算子についても同様です)．

このプログラムでは，ends マニピュレータを使用して，出力配列の終端を手作業でヌルにしています．配列の終端が自動的にヌルとされるかどうかは実装方法によって

異なるので，ヌル終端が重要となるアプリケーションでは常に手作業で行うのが最良の方法です．

**2.** 次のプログラムでは，配列ベースの入力を使用しています．

```
// 配列ベースの入力の使用例
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[] = "Hello 100 123.125 a";

  istrstream istr(buf); // 入力配列を開く

  int i;
  char str[80];
  float f;
  char c;

  istr >> str >> i >> f >> c;

  cout << str << ' ' << i << ' ' << f;
  cout << ' ' << c << '¥n';

  return 0;
}
```

このプログラムでは，入力配列 buf に含まれる値を読み取り，再び表示しています．

**3.** 入力配列は，いったんストリームにリンクすると，ファイルと同様に操作できることを覚えておいてください．たとえば，次のプログラムでは get() 関数と eof() 関数を使用して，buf の内容を読み取っています．

```
// 配列ベース入出力に対するget()関数とeof()関数の使用例
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[] = "Hello 100 123.125 a";

  istrstream istr(buf);
  char c;
```

```
  while(!istr.eof()) {
    istr.get(c);
    if(!istr.eof()) cout << c;
  }

  return 0;
}
```

**4.** 次のプログラムでは，配列に対して入出力を行ってます．

```
// 入出力配列の使用例
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iobuf[255];

  strstream iostr(iobuf, sizeof iobuf);

  iostr << "This is a test¥n";
  iostr << 100 << hex << ' ' << 100 << ends;

  char str[80];
  int i;

  iostr.getline(str, 79); // ¥nまでの文字列を読み取る
  iostr >> dec >> i; // 100を読み取る

  cout << str << ' ' << i << ' ';

  iostr >> hex >> i;
  cout <<  hex << i;

  return 0;
}
```

このプログラムでは，まずiobufに出力を書き込んでいます．次にそれを再び読み取ります．まず，getline()関数を使用して，

```
This is a test
```

という行全体を読み取っています．次に，10進数値100を読み取り，16進数値0x64を読み取っています．

**練習問題 13.7 配列ベースの入出力**

1. 例1のプログラムを修正し，終了前にbufに書き込まれた文字数を表示しなさい．
2. 配列ベースの入出力を使用して，1つの配列の内容を別の配列にコピーするアプリケーションを作成しなさい(言うまでもありませんが，これはあまり効率的な作業ではありません)．
3. 配列ベースの入出力を使用して，浮動少数点数を含む文字列を内部表現に変換するプログラムを作成しなさい．

# この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. staticメンバ関数がほかのメンバ関数と異なる点を説明しなさい．
2. 配列ベース入出力を使用する際にプログラムにインクルードするヘッダを挙げなさい．
3. メモリを入出力デバイスとして使うことのほかに，配列ベース入出力がC++の「通常の」入出力と異なる点があるかどうか述べなさい．
4. counter()という関数がある場合，この関数をC言語にリンクしてコンパイルするための文を示しなさい．
5. 変換関数の機能について説明しなさい．
6. explicitの目的を説明しなさい．
7. constメンバ関数に課されている基本的な制限とは何か説明しなさい．
8. 名前空間について説明しなさい．
9. mutableの機能について説明しなさい．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

1. 引数を1つしか必要としないコンストラクタは，引数の型をそのクラス型に自動的に変換しますが，この機能によって，オーバーロード代入演算子が不要になる状況があるかどうか説明しなさい．

2. constメンバ関数内でconst_castを使用し，呼び出し元オブジェクトの修正を許可することができるかどうか説明しなさい．

3. 最初のC++ライブラリはグローバルな名前空間に含まれており，古いC++プログラムはどれもこの状況に基づいて作成されています．ライブラリをstd名前空間に移動する利点を説明しなさい．

4. 第12章までに紹介したサンプルプログラムを見直し，constまたはstaticとして宣言できるメンバ関数があるかどうか考えなさい．また，名前空間を定義すべきサンプルプログラムがあるかどうか考えなさい．

# 14

# 標準テンプレートライブラリ（STL）

この章の内容

おめでとうございます！ これまでの章をすべて学習してきた方は，もう立派なC++プログラマです．この最後の章では，C++の最も興味深く高度な機能である，標準テンプレートライブラリについて説明します．

**標準テンプレートライブラリ**(Standard Template Library：**STL**)が組み込まれたことは，C++の標準化の過程において大きな功績でした．STLは最初のC++仕様には含まれていませんでしたが，標準化の過程で追加されました．STLには汎用のクラスと関数がテンプレート化されて含まれており，これらによってよく使われるアルゴリズムやデータ構造体が実装されています．たとえば，ベクトル，リスト，キュー，スタックなどが定義されています．また，それらにアクセスするための各種ルーチンも定義されています．STLはテンプレートクラスを基に作成されているので，そのアルゴリズムとデータ構造体は，ほとんどすべてのデータ型に適用することができます．

はじめに述べておかなければならないのは，STLは高度なC++機能を使用する複雑なソフトウェアエンジニアリングの産物であるということです．STLを理解して使用するには，これまでの章で説明した内容をすべて理解していなければなりません．特に，テンプレートについては熟知している必要があります．STLを記述するテンプレート構文は，(実際にはそれほど複雑ではありませんが)大変複雑に見えます．この章で紹介する内容には，本書でこれまでに学んできた内容よりも難しいものはありません．ただし，STLは一見複雑に見えるので驚かないでください．例を見て根気よく学習してください．見慣れない構文に惑わされて，STLの基本的な簡潔さを見失ってはいけません．

STLは大きなライブラリなので，そのすべての機能をこの章で説明することはできません．STLとその機能，詳細，プログラミング技法について説明しようとすれば，厚い本が1冊出来上がってしまいます．しかしここでは，STLの基本的な動作，設計方針，プログラミングの基盤に親しんでいただくために，概要を示します．この章に目を通せば，STLの残りの側面について，自分で学習できるようになるはずです．

この章では，C++の中でも特に重要な新しいクラスであるstringについても説明します．stringクラスは文字列データ型を定義します．これを使用すると，ほかのデータ型とほとんど同じように，演算子を使用して文字列を操作することができます．

## 前章の理解度チェック

この章を読み進む前に，次の問題に答えられるかどうか確認しましょう．正しく答えることができたら先へ進んでください．

**1.** C++に名前空間が追加された理由を説明しなさい．

**2.** const メンバ関数の指定方法を示しなさい.

**3.** 「mutable 修飾子を使用すると，ライブラリ関数をプログラムのユーザーが変更することができます.」この説明が正しいかどうかを答えなさい.

**4.** 次のクラスがあります.

```
class X {
  int a, b;
public:
  X(int i, int j) { a = i, b = j; }
  // ここにint型への変換関数を作成する
};
```

a と b の合計を返す整数変換関数を作成しなさい.

**5.** 「static メンバ関数は，そのクラスのオブジェクトを作成する前に使用することができます.」この説明が正しいかどうかを答えなさい.

**6.** 次のクラスがあります.

```
class Demo {
  int a;
public:
  explicit Demo(int i) { a = i; }
  int geta() { return a; }
};
```

次の宣言が有効かどうかを答えなさい.

```
Demo o = 10;
```

# 14.1　標準テンプレートライブラリの概要

標準テンプレートライブラリ(STL)は非常に大きく，その構文は複雑ですが，構造や要素を理解してしまえば，実際には大変使いやすいライブラリです．したがって，サンプルコードを見る前に，STLの概要を説明した方がよいでしょう．

標準テンプレートライブラリの中核には，基盤となる3つの要素があります．それはコンテナ，アルゴリズム，反復子です．これらを互いに組み合わせることによって，プログラミングに関するさまざまな問題を手軽な方法で解決することができます.

**コンテナ**(container)とは，ほかのオブジェクトを保持するオブジェクトのことです．コンテナにはいくつかの種類があります．たとえば，vectorクラスは動的配列を定義し，queueはキューを作成し，listは線形リストを提供します．基本的なコンテナに加えて，STLでは**連想コンテナ**(associative container)も定義されています．これを使用すると，キーを基にして値を効率的に取得することができます．たとえばmapクラスは，一意のキーを持つ値にアクセスするマップを定義します．したがって，マップにはキーと値の組み合わせが保存され，キーを指定して値を取得することができます．

各コンテナクラスでは，コンテナに対して使用できる一連の関数が定義されています．たとえば，リストコンテナには要素を挿入，削除，結合するための関数が含まれています．スタックには，値をプッシュおよびポップする関数が含まれています．

**アルゴリズム**(algorithm)はコンテナを処理します．アルゴリズムが行うサービスとしては，コンテナの内容の初期化，ソート，検索，変形などがあります．多くのアルゴリズムはシーケンス(コンテナ内の要素の線形リスト)を処理します．

**反復子**(iterator)はオブジェクトで，ポインタと似ています．反復子を使うと，ポインタを使って配列を循環処理するのと同じ方法で，コンテナの内容を循環処理することができます．次の表に，5種類の反復子を示します．

**表14-1　反復子の種類**

| 反復子 | 許可されているアクセス権 |
|---|---|
| ランダムアクセス反復子 | 値を格納および検索する．要素にはランダムにアクセスできる |
| 双方向反復子 | 値を格納および検索する．前方または後方に移動する |
| 前方反復子 | 値を格納および検索する．前方にのみ移動する |
| 入力反復子 | 値を検索する．格納はしない．前方にのみ移動する |
| 出力反復子 | 値を格納する．検索はしない．前方にのみ移動する |

一般的には，多くのアクセス機能を持つ反復子は，それよりもアクセス機能が少ない反復子の代わりとして使用することができます．たとえば，前方反復子は，入力反復子の代わりとして使うことができます．

反復子はポインタと同様に扱うことができます．反復子をインクリメントまたはデクリメントすることができます．また，*演算子を適用することもできます．反復子を宣言するには，各種コンテナによって定義されている**iterator**型を使用します．

STLでは，逆反復子もサポートされています．逆反復子は双方向またはランダムアクセスの反復子で，シーケンス内を逆方向に移動します．したがって，逆反復子がシーケンスの最後を指している場合，その反復子をインクリメントすると，最後から1つ前の要素を指すようになります．

テンプレートに関する説明の中でさまざまな反復子型に言及する際，本書では次の表に示す用語を使用します．

**表14-2　反復子に関する本書での用語**

| 用語 | 反復型 |
|---|---|
| BiIter | 双方向反復子 |
| ForIter | 前方反復子 |
| InIter | 入力反復子 |
| OutIter | 出力反復子 |
| RandIter | ランダムアクセス反復子 |

コンテナ，アルゴリズム，反復子に加えて，STLのサポートはその他のいくつかの標準コンポーネントに依存しています．その中で主なものとしては，アロケータ，条件式，比較関数があります．

各コンテナには専用のアロケータが定義されています．アロケータはコンテナのメモリ割り当てを管理します．デフォルトのアロケータはallocatorクラスのオブジェクトですが，特殊なアプリケーションで必要に応じて独自のアロケータを定義することもできます．ただし，ほとんどの場合はデフォルトのアロケータで十分です．

一部のアルゴリズムとコンテナでは，**条件式**と呼ばれる特殊な関数を使用します．条件式には単項と2項の2種類があります．**単項条件式**は引数を1つ受け取り，**2項条件式**は引数を2つ受け取ります．これらの関数は真または偽を返します．真と偽のどちらを返すかを決める正確な条件はプログラマが定義します．この章では，単項条件式関数を使用するときは，UnPred型を使用します．2項条件式関数を使用するときは，BinPred型を使用します．2項条件式では，引数は常に第1引数，第2引数の順序で渡されます．単項条件式と2項条件式のどちらの場合でも，引数にはコンテナによって格納されるオブジェクトと同じ型の値が含まれています．

一部のアルゴリズムとクラスでは，2つの要素を比較する特殊な2項条件式を使用します．この種の条件式は比較関数と呼ばれ，1つ目の引数が2つ目の引数よりも小さい場合に真を返します．比較関数にはComp型を使用します．

各種STLクラスで必要とされるヘッダに加えて，C++標準ライブラリにはSTLのサポートを提供する<utility>ヘッダと<functional>ヘッダも含まれています．たとえば，<utility>には，1組の値を格納することのできるテンプレートクラスpairの定義が含まれています．この章でも後から実際にpairを使用します．

<functional>に含まれるテンプレートを使用すると，operator()を定義するオブジェクトを作成することができます．これらは**関数オブジェクト**と呼ばれ，多くの状況で関数ポインタの代わりとして使うことができます．<functional>ではいくつかの定義済み関数オブジェクトが宣言されています．その一部を次に示します．

**表14-3　定義済み関数オブジェクトの一部**

| | | | | |
|---|---|---|---|---|
| plus | minus | multiplies | divides | modulus |
| negate | equal_to | not_equal_to | greater | greater_equal |
| less | less_equal | logical_and | logical_or | logical_not |

最も広く使われている関数オブジェクトはおそらくlessです．lessを使うと，あるオブジェクトの値がほかのオブジェクトの値よりも小さいかどうかを調べることができます．後述するSTLアルゴリズムでは，実際の関数ポインタの代わりとして関数オブジェクトを使用することができます．関数ポインタの代わりに関数オブジェクトを使用すると，STLでより効率的なプログラムコードを生成することができます．ただしこの章では，関数オブジェクトは必要ないので，直接使うことはありません．関数オブジェクトは難しくありませんが，関数オブジェクトについて詳しく説明すると非常に長くなり，本書の主題の範囲を超えてしまいます．ただ，STLの効率を最大限に引き出すために必要な機能であることを覚えておいてください．

**練習問題　14.1　標準テンプレートライブラリの概要**

1. STLに関するコンテナ，アルゴリズム，反復子について説明しなさい．
2. 2種類の条件式を挙げなさい．
3. 5種類の反復子を挙げなさい．

# 14.2　コンテナクラス

前述のとおり，**コンテナ**(container)とは実際にデータを格納するSTLオブジェクトのことです．STLによって定義されているコンテナを表14-4に示します．また，各コンテナを使用するためにインクルードするヘッダも示します．文字列を管理するstringクラスもコンテナですが，このクラスについてはこの章の後半で説明します．

テンプレートクラス宣言内のプレースホルダ型の名前は任意なので，コンテナクラスではこれらの型のtypedef版を宣言しています．これによって，型名が具体的になります．表14-5に，一般的に使われるtypedef名を示します．

**表 14-4　STL によって定義されているコンテナ**

| コンテナ | 説明 | 必要なヘッダ |
|---|---|---|
| bitset | ビットのセット | <bitset> |
| deque | double 終了キュー | <deque> |
| list | 線形リスト | <list> |
| map | キーと値の組み合わせを保存する．各キーには値が 1 つだけ関連付けられる | <map> |
| multimap | キーと値の組み合わせを保存する．各キーに 2 つ以上の値を関連付けることができる | <map> |
| multiset | 各要素が一意とは限らないセット | <set> |
| priority_queue | 優先度キュー | <queue> |
| queue | キュー | <queue> |
| set | 各要素が一意であるセット | <set> |
| stack | スタック | <stack> |
| vector | 動的配列 | <vector> |

**表 14-5　コンテナクラスで宣言する typedef 名(よく使われるもの)**

| typedef 名 | 説明 |
|---|---|
| size_type | size_t と同等な整数型 |
| reference | 要素の参照 |
| const_reference | 要素の const 参照 |
| iterator | 反復子 |
| const_iterator | const 反復子 |
| reverse_iterator | 逆反復子 |
| const_reverse_iterator | const 逆反復子 |
| value_type | コンテナに格納される値の型 |
| allocator_type | アロケータの型 |
| key_type | キーの型 |
| key_compare | 2 つのキーを比較する関数の型 |
| value_compare | 2 つの値を比較する関数の型 |

この章では，これらの各コンテナについて詳しく説明することはできませんが，次節では代表的な3つのコンテナである，ベクトル，リスト，マップについて説明します．これらのコンテナの動作を理解すれば，ほかのコンテナも問題なく使用できるはずです．

# 14.3　ベクトル

コンテナの最も一般的な使用目的は，おそらく**ベクトル**(vector) でしょう．vectorクラスは動的配列をサポートしています．動的配列とは，必要に応じてサイズを増やすことのできる配列のことです．ご存じのとおり，C++では配列のサイズはコンパイル時に固定されています．これは配列を実装する上で非常に効率的な方法ではありますが，実行時にプログラムの状態の変化に応じて配列のサイズを調整することができないので，最も制限が厳しい方法でもあります．ベクトルは，メモリを必要に応じて割り当てることによって，この問題を解決します．ベクトルは動的ですが，やはり標準的な配列添え字表記を使用してその要素にアクセスすることができます．

ベクトルのテンプレート指定を次に示します．

**ベクトルのテンプレート指定**

```
template <class T, class Allocator = allocator<T>> class vector
```

Tには格納するデータの型を指定し，Allocatorにはアロケータを指定します．デフォルトのアロケータは標準アロケータです．vectorクラスには，次のコンストラクタがあります．

**vector クラス**

```
explicit vector(const Allocator &a = Allocator( ) );
explicit vector(size_type num, const T &val = T ( ),
                    const Allocator &a = Allocator( ));
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
                    const Allocator &a = Allocator( ));
```

1つ目の形式では，空のベクトルを作成します．2つ目の形式では，値*val*を持つ*num*個の要素を含むベクトルを作成します．値*val*には，デフォルト値を使用することができます．3つ目の形式では，*ob*と同じ要素を含むベクトルを作成します．4つ目の形式では，反復子*start*と*end*によって指定した範囲に含まれる要素を持つベクトルを作成します．

ベクトルに保存されるすべてのオブジェクトには，デフォルトのコンストラクタを定義する必要があります．また，<および==操作も定義する必要があります．一部のコンパイラでは，その他の比較演算子の定義も必須とされています(正確な情報はコンパイラの実装方法によって異なるので，使用するコンパイラのマニュアルを参照してください)．組み込みの型はすべてこれらの要件を自動的に満たしています．

テンプレートの構文は一見複雑に見えますが，ベクトルの宣言は難しくありません．次に例を示します．

```
vector<int> iv;             // 長さゼロのintベクトルを作成する
vector<char> cv(5);         // 5要素のcharベクトルを作成する
vector<char> cv(5, 'x');    // 5要素のcharベクトルを初期化する
vector<int> iv2(iv);        // intベクトルを基にintベクトルを作成する
```

vector クラスには，次の比較演算子が定義されています．

**vector クラスの比較演算子**

```
==, <, <=, !=, >, >=
```

また，vector クラスには添え字演算子[ ]も定義されています．添え字演算子を使用すると，標準的な配列添え字表記を使用してベクトルの要素にアクセスすることができます．

vector クラスで定義されているメンバ関数を表14-6に示します(構文を見て圧倒されないようにしましょう)．中でも重要なメンバ関数は，size()，begin()，end()，push_back()，insert()，erase()です．size()関数はベクトルの現在のサイズを返します．この関数を使うと，ベクトルのサイズを実行時に調べることができるので，大変便利です．ベクトルのサイズは必要に応じて増えることを覚えておいてください．したがって，ベクトルのサイズはコンパイル中ではなく実行時に判別しなければなりません．

**表14-6　vector クラスのメンバ関数**

| メンバ関数 | 説明 |
|---|---|
| template <class InIter><br>void **assign**(InIter *start*, InIter *end*); | *start*と*end*によって定義されるシーケンスをベクトルに代入する |
| template <class Size, class T><br>void **assign**(Size *num*, const T &*val* = T( )); | 値*val*を持つベクトル*num*要素を代入する |
| reference **at**(size_type *i* );<br>const_reference **at**(size_type *i* ) const; | *i*に指定した要素への参照を返す |
| reference **back**( );<br>const_reference **back**( ) const; | ベクトル内の最後の要素への参照を返す |
| iterator **begin**( );<br>const_iterator **begin**( ) const; | ベクトルの最初の要素を指す反復子を返す |
| size_type **capacity**( ) const; | ベクトルの現在のサイズ(割り当てるメモリを増やすことなく格納できる要素の数)を返す |
| void **clear**( ); | ベクトルのすべての要素を削除する |

(続く)

（続き）

| メンバ関数 | 説明 |
|---|---|
| bool **empty**( ) const; | 呼び出し元のベクトルが空の場合は真，空ではない場合は偽を返す |
| iterator **end**( );<br>const_iterator **end**( ) const; | ベクトルの末尾を指す反復子を返す |
| iterator **erase**(iterator *i* ); | *i*が指す要素を削除する．削除した要素の後を指す反復子を返す |
| iterator **erase**(iterator *start*, iterator *end* ); | *start*から*end*までの範囲の要素を削除する．削除した最後の要素の後の要素を指す反復子を返す |
| reference **front**( );<br>const_reference **front**( ) const; | ベクトル内の最初の要素への参照を返す |
| allocator_type **get_allocator**( ) const; | ベクトルのアロケータを返す |
| iterator **insert**(iterator *i*, const T &*val* = T( )); | *i*に指定した要素の直前に*val*を挿入する．この要素を指す反復子を返す |
| void **insert**(iterator *i*, size_type *num*, const T &*val* ) | *i*に指定した要素の直前に *num* 個の値 *val* を挿入する |
| template <class InIter><br>void **insert**(iterator *i*, InIter *start*, InIter *end* ); | *i*に指定した要素の直前に *start* と *end* によって定義されるシーケンスを挿入する |
| size_type **max_size**( ) const; | ベクトルに格納できる要素の最大数を返す |
| reference **operator[ ]**(size_type *i* ) const;<br>const_reference **operator[ ]**(size_type *i* ) const; | *i*に指定した要素への参照を返す |
| void **pop_back**( ); | ベクトル内の最後の要素を削除する |
| void **push_back**(const T &*val* ); | *val*に指定した値を持つ要素をベクトルの末尾に追加する |
| reverse_iterator **rbegin**( );<br>const_reverse_iterator **rbegin**( ) const; | ベクトルの末尾を指す逆反復子を返す |
| reverse_iterator **rend**( );<br>const_reverse_iterator **rend**( ) const; | ベクトルの先頭を指す逆反復子を返す |
| void **reserve**(size_type *num*); | ベクトルのサイズが *num* 以上になるように設定する |
| void **resize**(size_type *num*, T *val* = T ( )); | ベクトルのサイズを*num*に指定したサイズに変更する．ベクトルのサイズを増やす場合は，*val* に指定した値を持つ要素を末尾に追加する |
| size_type **size**( ) const; | ベクトルに現在含まれている要素の数を返す |
| void **swap**(vector<T, Allocator> &*ob*) | 呼び出し元ベクトルに含まれている要素を *ob* 内の要素と交換する |

begin()関数は，ベクトルの先頭を指す反復子を返します．end()関数はベクトルの末尾を指す反復子を返します．前述のとおり，反復子はポインタに似ており，begin()関数とend()関数を使用して，ベクトルの先頭と末尾を指す反復子を取得することができます．

push_back()関数は，ベクトルの末尾に値を追加します．必要であれば，新しい要素を含めるためにベクトルのサイズが増やされます．insert()関数を使うと，ベクトルの中間に要素を追加することができます．また，ベクトルを初期化することもできます．どのような場合でも，ベクトルに要素を含めた後は，配列添え字を使用してそれらの要素を取得または修正することができます．ベクトルから要素を削除するには，erase()関数を使用します．

## 例 14.3　ベクトル

**1.** 次の短いプログラムでは，ベクトルの基本的な操作方法を示しています．

```
// ベクトルの基本操作
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // 長さゼロのベクトルを作成する
  int i;

  // vの最初のサイズを表示する
  cout << "Size = " << v.size() << endl;

  /* ベクトルの末尾に値を追加する
     必要に応じてベクトルのサイズが増える */
  for(i=0; i<10; i++) v.push_back(i);

  // vの現在のサイズを表示する
  cout << "Size now = " << v.size() << endl;

  // ベクトルの内容を表示する
  cout << "Current contents:¥n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  /* ベクトルの末尾にさらに値を追加する
     この場合も，必要に応じてベクトルのサイズが増える */
  for(i=0; i<10; i++) v.push_back(i+10);

  // vの現在のサイズを表示する
```

```
    cout << "Size now = " << v.size() << endl;

    // ベクトルの内容を表示する
    cout << "Current contents:¥n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    // ベクトルの内容を変更する
    for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

    // ベクトルの内容を表示する
    cout << "Contents doubled:¥n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    return 0;
  }
```

このプログラムからの出力を次に示します．

```
Size = 0
Size now = 10
Current contents:
0 1 2 3 4 5 6 7 8 9
Size now = 20
Current contents:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Contents doubled:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

このプログラムを詳しく見ていくことにしましょう．main()関数では，vという名前の整数ベクトルを作成しています．初期化を行っていないので，これは初期サイズがゼロの空のベクトルになります．つまり，長さゼロのベクトルです．このプログラムではsize()メンバ関数を呼び出すことによって，このことを確認しています．次に，メンバ関数push_back()を使用して，vの末尾に10個の要素を追加しています．これによって，新しい要素を収めるためにvのサイズが増えます．出力内容からもわかるように，この操作の後，vのサイズは10になります．次に，vの内容を出力します．ここでは，標準配列添え字表記を使用している点に注目してください．次に，さらに10個の要素をvに追加します．vのサイズはこれに合わせて自動的に増やされます．最後に標準配列添え字表記を使用して，vの要素値を変更します．

このプログラムには，興味深い点が1つあります．vの内容を表示するループでは，ターゲットとしてv.size()を使用している点に注目してください．配列と比べたときのべ

クトルの利点の1つとして，ベクトルの現在のサイズを調べることができるという点が挙げられます．この機能はさまざまな状況で役に立ちます．

**2.** ご存じのとおり，C++ の配列とポインタには密接なつながりがあります．配列には，添え字とポインタのどちらを使用してもアクセスすることができます．STLにおいては，ベクトルと反復子の間にも同じ関係があります．ベクトルのメンバには，添え字を使用しても，また反復子を使用してもアクセスできます．次の例では，これらの両方の手法を使用しています．

```
// 反復子を使用してベクトルにアクセスする
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // 長さゼロのベクトルを作成する
  int i;

  // ベクトルに値を追加する
  for(i=0; i<10; i++) v.push_back(i);

  // 添え字を使用してベクトルのないようにアクセスできる
  for(i=0; i<10; i++) cout << v[i] << " ";
  cout << endl;

  // 反復子を使用してアクセスする
  vector<int>::iterator p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

このプログラムからの出力を次に示します．

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

このプログラムでは，最初に長さゼロのベクトルを作成します．次に，push_back()メンバ関数を使用してベクトルの最後に値を追加し，必要に応じてサイズを増やします．

反復子pの宣言方法に注目してください．コンテナクラスによって，iterator型が定義されています．したがって，特定のコンテナの反復子を取得するには，例に示したような宣言を使用します．つまり，反復子に単にコンテナの名前を付けます．このプログラムでは，begin()メンバ関数を使用して，ベクトルの先頭を指すようにpを初期化しています．この関数は，ベクトルの先頭を指す反復子を返します．この反復子を必要に応じてインクリメントすれば，ベクトルの要素に1つずつアクセスすることができます．この作業は，ポインタを使って配列の要素にアクセスする方法とよく似ています．ベクトルの最後に達したかどうかを判別するには，end()メンバ関数を使用します．この関数はベクトル内の最後の要素の直後の位置を指す反復子を返します．したがって，pがv.end()に一致した場合にはベクトルの最後に達したことがわかります．

**3.** ベクトルの末尾に新しい値を追加するだけではなく，insert()関数を使用すると，中間に要素を挿入することができます．また，erase()関数を使用すると要素を削除することができます．次のプログラムは，insert()関数とerase()関数の使用例を示しています．

```
// insert()関数とerase()関数の使用例
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(5, 1); // 値1を含む5要素のベクトルを作成する
  int i;

  // ベクトルの最初の内容を表示する
  cout << "Size = " << v.size() << endl;
  cout << "Original contents:¥n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  vector<int>::iterator p = v.begin();
  p += 2; // 3つ目の要素を指す

  // 値9を持つ10個の要素を挿入する
  v.insert(p, 10, 9);

  // 挿入後のベクトルの内容を表示する
  cout << "Size after insert = " << v.size() << endl;
  cout << "Contents after insert:¥n";
```

```
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  // これらの要素を削除する
  p = v.begin();
  p += 2; // 3つ目の要素を指す
  v.erase(p, p+10); // 次の10個の要素を削除する

  // 削除後のベクトルの内容を表示する
  cout << "Size after erase = " << v.size() << endl;
  cout << "Contents after erase:¥n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Size = 5
Original contents:
1 1 1 1 1
Size after insert = 15
Contents after insert:
1 1 9 9 9 9 9 9 9 9 9 9 1 1 1
Size after erase = 5
Contents after erase:
1 1 1 1 1
```

**4.** 次のプログラムでは，ベクトルを使用してプログラマ定義クラスのオブジェクトを格納しています．このクラスではデフォルトのコンストラクタを定義し，<と==のオーバーロードバージョンを使用しています．使用するコンパイラにおけるSTLの実装方法によっては，ほかの比較演算子も定義しなければならないことがあるので，注意してください．

```
// ベクトルにクラスオブジェクトを保存する
#include <iostream>
#include <vector>
using namespace std;

class Demo {
  double d;
public:
  Demo() { d = 0.0; }
  Demo(double x) { d = x; }
  Demo &operator=(double x) {
    d = x; return *this;
```

```
  }
  double getd() { return d; }
};

bool operator<(Demo a, Demo b)
{
  return a.getd() < b.getd();
}

bool operator==(Demo a, Demo b)
{
  return a.getd() == b.getd();
}

int main()
{
  vector<Demo> v;
  int i;

  for(i=0; i<10; i++)
    v.push_back(Demo(i/3.0));

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " ";

  cout << endl;

  for(i=0; i<v.size(); i++)
    v[i] = v[i].getd() * 2.1;

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " ";

  return 0;
}
```

このプログラムからの出力を次に示します.

```
0 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3
```

## 練習問題 14.3 ベクトル

**1.** 例に示した各プログラムに小さな変更を加え，その結果を観察しなさい.

**2.** 例4では，Demoクラスにデフォルト(仮引数なしの)コンストラクタと仮引数付きのコンストラクタを両方とも用意しました．この手法が重要である理由を説明しなさい．

**3.** 次に，単純なCoordクラスを示します．Coord型のオブジェクトをベクトルに保存するプログラムを作成しなさい．

ヒント　Coordに対する < 演算子と == 演算子を定義するのを忘れないようにしましょう．

```
class Coord {
public:
  int x, y;
  Coord() { x = y = 0; }
  Coord(int a, int b) { x = a; y = b; }
};
```

# 14.4　リスト

listクラスは双方向の線形リストをサポートしています．ランダムアクセスをサポートするベクトルとは異なり，リストには順次アクセスしかできません．リストは双方向なので，前方から後方へも，後方から前方へのアクセスすることができます．

listクラスのテンプレート指定を次に示します．

**リストのテンプレート指定**

```
template <class T, class Allocator = allocator<T>> class list
```

Tには，リストに保存するデータの型を指定します．Allocatorにはアロケータを使用します．デフォルトのアロケータは標準アロケータです．listクラスには次のコンストラクタがあります．

**listクラス**

```
explicit list(const Allocator &a = Allocator( ) );
explicit list(size_type num, const T &val = T ( ),
              const Allocator &a = Allocator( ));
list(const list<T, Allocator> &ob);
template <class InIter> list(InIter start, InIter end,
                             const Allocator &a = Allocator( ));
```

1つ目の形式では，空のリストを作成します．2つ目の形式では，値*val*を持つ*num*個の要素を含むリストを作成します．値*val*にはデフォルト値を使用することもできます．3つ目の形式では，*ob*と同じ要素を含むリストを作成します．4つ目の形式では，反復子*start*と*end*によって指定した範囲の要素を含むリストを作成します．

listクラスでは，次の比較演算子が定義されています．

**listクラスの比較演算子**

```
==, <, <=, !=, >, >=
```

listクラスに定義されているメンバ関数を表14-7に示します．ベクトルと同様，リストに値を追加するには，push_back()関数を使用します．リストの先頭に要素を追加するには，push_front()関数を使用します．リストの中間に要素を挿入するには，insert()関数を使用します．2つのリストを組み合わせるにはsplice()関数を使用し，1つのリストをほかのリストに結合するにはmerge()関数を使用します．

**表14-7　listクラスのメンバ関数**

| メンバ関数 | 説明 |
|---|---|
| template <class InIter><br>void **assign**(InIter *start*, InIter *end* ); | *start*と*end*によって定義されるシーケンスをリストに代入する |
| template <class Size, class T><br>void **assign**(Size *num*,<br>const T &*val* = T( )); | 値*val*を持つ*num*個の要素をリストに代入する |
| reference **back**( );<br>const_reference **back**( ) const; | リスト内の最後の要素への参照を返す |
| iterator **begin**( );<br>const_iterator **begin**( ) const; | リスト内の最初の要素を指す反復子を返す |
| void **clear**( ); | リストからすべての要素を削除する |
| bool **empty**( ) const; | 呼び出し元リストが空の場合は真，空ではない場合は偽を返す |
| iterator **end**( );<br>const_iterator **end**( ) const; | リストの末尾を指す反復子を返す |
| iterator **erase**(iterator *i* ); | *i*が指す要素を削除する．削除した要素の直後の要素を指す反復子を返す |
| iterator **erase**(iterator *start*, iterator *end* ); | *start*から*end*までの範囲の要素を削除する．削除した最後の要素の直後の要素を指す反復子を返す |

| メンバ関数 | 説明 |
|---|---|
| reference **front**( );<br>const_reference **front**( ) const; | リスト内の最初の要素への参照を返す |
| allocator_type **get_allocator**( ) const; | リストのアロケータを返す |
| iterator **insert**(iterator *i*,<br>const T &*val* = T( )); | *i*に指定した要素の直前に*val*を挿入する．その要素を指す反復子を返す |
| void **insert**(iterator *i*, size_type *num*,<br>const T &*val* ) | *i*に指定した要素の直前に*num*個の値*val*を挿入する |
| template <class InIter><br>void **insert**(iterator *i*<br>InIter *start*, InIter *end* ); | *i*に指定した要素の直前に*start*と*end*によって定義されるシーケンスを挿入する |
| size_type **max_size**( ) const; | リストに格納できる要素の最大数を返す |
| void **merge**(list<T, Allocator> &*ob*);<br>template <class Comp><br>void **merge**(<list<T, Allocator> &*ob*,<br>Comp *cmpfn*); | *ob*に含まれる順序付きリストを，呼び出し元順序付きリストに結合する．結果は順序付きになる．結合の後，*ob*に含まれるリストは空になる．2つ目の形式では，比較関数を指定して，1つの要素の値がほかの要素の値よりも小さいかどうかを判別することができる |
| void **pop_back**( ); | リスト内の最後の要素を削除する |
| void **pop_front**( ); | リスト内の最初の要素を削除する |
| void **push_back**(const T &*val* ); | *val*に指定した値を持つ要素をリストの最後に追加する |
| void **push_front**(const T &*val* ); | *val*に指定した値を持つ要素をリストの先頭に追加する |
| reverse_iterator **rbegin**( );<br>const_reverse_iterator **rbegin**( ) const; | リストの末尾を指す逆反復子を返す |
| void **remove**(const T &*val* ); | 値*val*を持つ要素をリストをから返す |
| template <class UnPred><br>void **remove_if**(UnPred *pr*); | 単項条件式*pr*が真である要素を削除する |
| reverse_iterator **rend**( ); | リストの先頭を指す逆反復子を返す |
| const_reverse_iterator **rend**( ) const; | リストのサイズを*num*に指定したサイズに変更する．リストのサイズを増やす場合は，*val*に指定した値を持つ要素が末尾に追加される |
| void **resize**(size_type *num*, T val = T ( )); | 呼び出し元リストを逆順にする |
| void **reverse**( );<br>size_type **size**( ) const; | リストに現在含まれている要素の数を返す |
| void **sort**( );<br>template <class Comp><br>void **sort**(Comp *cmpfn*); | リストをソートする．2つ目の形式では，比較関数*cmpfn*を使用して，2つの要素値の大小を判別することによって，リストをソートする |
| void **splice**(iterator *i*,<br>list<T, Allocator> &*ob*); | *ob*の内容を呼び出し元リストの位置*i*に挿入する．操作後，*ob*は空になる |

（続く）

（続き）

| メンバ関数 | 説明 |
|---|---|
| void **splice**(iterator *i*,<br>list<T, Allocator> *&ob*,<br>iterator *el* ); | *el* が指す要素をリスト *ob* から削除し，それを呼び出し元リストの位置 *i* に保存する |
| void **splice**(iterator *i*,<br>list<T, Allocator> *&ob*,<br>iterator *start*, iterator *end* ); | *start* と *end* によって定義される範囲を *ob* から削除し，それを呼び出し元リストの *i* を開始位置として保存する |
| void **swap**(list< T, Allocator> *&ob*) | 呼び出し元リストに含まれる要素を *ob* の要素と交換する |
| void **unique**( );<br>template <class BinPred><br>void **unique**(BinPred *pr* ); | 呼び出し元リストから重複する要素を削除する．２つ目の形式では，*pr* を使用して一意性を判別する |

リストに保存するすべてのデータ型には，デフォルトのコンストラクタを定義する必要があります．また，各種比較演算子も定義しなければなりません．本書の執筆時点では，リストに保存されるオブジェクトの正確な要件はコンパイラによって異なり，将来，変更される可能性もあります．そのため，詳細については使用するコンパイラのマニュアルを参照してください．

## 例 14.4　リスト

**1.** 次に，リストの単純な例を示します．

```
// リストの基本操作
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst; // 空のリストを作成する
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size = " << lst.size() << endl;

  list<char>::iterator p;

  cout << "Contents: ";
  while(!lst.empty()) {
    p = lst.begin();
```

```
    cout << *p;
    lst.pop_front();
  }

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Size = 10
Contents: ABCDEFGHIJ
```

このプログラムでは，文字のリストを作成しています．まず，空のlistオブジェクトを作成します．次に，10個の文字(AからJまで)をリストに追加します．このためにはpush_back()関数を使用します．この関数を使用すると，既存のリストの末尾に新しい値を追加することができます．次に，リストのサイズを表示します．そして，リストの内容を繰り返し取得して表示し，リスト内の最初の要素を削除することによって出力します．リストが空になると，この処理が終了します．

**2.** 例1では，リストをループ処理しながら空にしました．これは，当然ながら必須の処理ではありません．たとえば，リストの内容を表示するループを次のように修正することもできます．

```
list<char>::iterator p = lst.begin();

while(p != lst.end()) {
  cout << *p;
  p++;
}
```

ここでは，反復子pをリストの先頭に初期化しています．ループを通るたびにpをインクリメントし，次の要素を指すようにしています．pがリストの最後に達すると，ループが終了します．

**3.** リストは双方向なので，リストの前方からでも後方からでも要素を追加することができます．たとえば，次のプログラムでは2つのリストを作成し，1つ目のリストを逆順にして2つ目のリストとしています．

```
// 要素をリストの前方からでも後方からでも追加することができる
#include <iostream>
#include <list>
using namespace std;

int main()
```

```
{
  list<char> lst;
  list<char> revlst;
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size of lst = " << lst.size() << endl;
  cout << "Original contents: ";

  list<char>::iterator p;

  /* lstから要素を削除し，その要素を
     revlstに逆順に追加する */
  while(!lst.empty()) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
    revlst.push_front(*p);
  }
  cout << endl << endl;

  cout << "Size of revlst = ";
  cout << revlst.size() << endl;
  cout << "Reversed contents: ";
  p = revlst.begin();
  while(p != revlst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Size of lst = 10
Original contents: ABCDEFGHIJ
Size of revlst = 10
Reversed contents: JIHGFEDCBA
```

このプログラムでは，lstの先頭から要素を削除し，それをrevlstの先頭から追加していくことによって，リストを逆順にしています．これによって，revlstには要素が逆順で保存されます．

**4.** sort()メンバ関数を使用すると，リストをソートすることができます．次のプログラムでは，無作為の文字を含むリストを作成し，リストの要素をソートしています．

```
// リストをソートする
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<char> lst;
  int i;

  // 無作為の文字を含むリストを作成する
  for(i=0; i<10; i++)
    lst.push_back('A'+ (rand()%26));

  cout << "Original contents: ";
  list<char>::iterator p = lst.begin();

  while(p != lst.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // リストをソートする
  lst.sort();

  cout << "Sorted contents: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

このプログラムからの出力を次に示します.

```
Original contents: PHQGHUMEAY
Sorted contents: AEGHHMPQUY
```

5. 順序付きリストを，別の順序付きリストと結合することができます．その結果，元の2つのリストの内容を含む順序付きリストが作成されます．結合した内容は呼び出し元リストに格納され，もう一方のリストは空になります．この例では2つのリストを結合します．1つ目のリストには文字ACEGI，2つ目のリストには文字BDFHJが含

まれています．次にこれらのリストを結合し，ABCDEFGHIJというシーケンスを作成します．

```
// 2つのリストを結合する
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst1, lst2;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Contents of lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;
  cout << "Contents of lst2: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // 2つのリストを結合する
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "lst2 is now empty¥n";
  cout << "Contents of lst1 after merge:¥n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Contents of lst1: ACEGI
Contents of lst2: BDFHJ
lst2 is now empty
Contents of lst1 after merge:
ABCDEFGHIJ
```

**6.** 次のプログラムでは，リストを使ってProject型オブジェクトを保存しています．Projectクラスは，ソフトウェアプロジェクトの管理に役立つクラスです．<，>, !=, ==の各演算子をProject型オブジェクト用にオーバーロードしている点に注目してください．これらはMicrosoft Visual C++ 5.0（この章のSTLのサンプルプログラムをテストするために使用したコンパイラ）で必須とされる演算子です．コンパイラによっては，このほかの演算子もオーバーロードしなければならない場合があります．STLではこれらの関数を使用して，コンテナ内のオブジェクトの順序を調べています．リストは順序付きコンテナではありませんが，検索，ソート，結合を行う際には，やはり要素を比較する手段が必要となります．

```
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
  char name[40];
  int days_to_completion;
  Project() {
    strcpy(name, "");
    days_to_completion = 0;
  }
  Project(char *n, int d) {
    strcpy(name, n);
    days_to_completion = d;
  }

  void add_days(int i) {
    days_to_completion += i;
  }

  void sub_days(int i) {
    days_to_completion -= i;
  }

  bool completed() { return !days_to_completion; }

  void report() {
```

```
    cout << name << ": ";
    cout << days_to_completion;
    cout << " days left.¥n";
  }
};

bool operator<(const Project &a, const Project &b)
{
  return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
  return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
  return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
  return a.days_to_completion != b.days_to_completion;
}

int main()
{
  list<Project> proj;

  proj.push_back(Project("Compiler", 35));
  proj.push_back(Project("Spreadsheet", 190));
  proj.push_back(Project("STL Implementation", 1000));
  list<Project>::iterator p = proj.begin();

  /* プロジェクトを表示する */
  while(p != proj.end()) {
    p->report();
    p++;
  }

  // 1つ目のプロジェクトに10日を追加する
  p = proj.begin();
  p->add_days(10);

  // 1つ目のプロジェクトを完了まで移動する
  do {
    p->sub_days(5);
```

```
    p->report();
  } while (!p->completed());

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Compiler: 35 days left.
Spreadsheet: 190 days left.
STL Implementation: 1000 days left.
Compiler: 40 days left.
Compiler: 35 days left.
Compiler: 30 days left.
Compiler: 25 days left.
Compiler: 20 days left.
Compiler: 15 days left.
Compiler: 10 days left.
Compiler: 5 days left.
Compiler: 0 days left.
```

## 練習問題 14.4 リスト

**1.** 例に小さな変更を加え，実験しなさい．

**2.** 例1では，リストの内容を表示しながらリストを空にしました．例2では，内容を削除しない方法でリストをループ処理しました．このほかに，リストを空にすることなくループ処理する方法を考えなさい．例1のプログラムを修正して，その方法を確認しなさい．

**3.** 例6のプログラムを拡張し，次の値を含むプロジェクトリストを追加しなさい．

| プロジェクト | 完了日数 |
| --- | --- |
| Database | 780 |
| Mail merge | 50 |
| COM objects | 300 |

次に，2つのリストをソートし，結合しなさい．また，最終的な結果を表示しなさい．

## 14.5　マップ

mapクラスは連想コンテナをサポートします．連想コンテナでは，一意のキーが値に対応付けられます．キーとは，値に付ける単なる名前のようなものです．値を保存したら，キーを使ってそれを取得することができます．したがって，大きな意味では，マップとはキーと値の組み合わせのリストであると考えることができます．マップの利点は，キーを使って値を検索できるという点です．たとえば，人名をキーとして使用し，その人物の電話番号を値として保存するマップを定義することができます．連想コンテナは，プログラミングにおいてますます広く使われるようになっています．

前述のとおり，マップには一意のキーしか含めることができません．重複したキーを使うことはできません．重複したキーを使用できるマップを作成するには，**multimap**を使用します．

mapコンテナのテンプレート指定を次に示します．

**マップのテンプレート指定**

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<T>> class map
```

Keyにはキーのデータ型を指定します．Tには保存する(対応付ける)値のデータ型を指定します．Compには2つのキーを比較する関数を指定します．Compのデフォルト値は標準less()ユーティリティ関数オブジェクトです．Allocatorには，アロケータを指定します(デフォルト値はallocatorです)．

mapクラスには，次のコンストラクタがあります．

**mapクラス**

```
explicit map(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ) );
map(const map<Key, T, Comp, Allocator> &ob);
template <class InIter> map(InIter start, InIter end,
  const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));
```

1つ目の形式では空のマップを作成します．2つ目の形式では，*ob*と同じ要素を含むマップを作成します．3つ目の形式では，反復子*start*と*end*によって指定した範囲の要素を含むマップを作成します．*cmpfn*に関数を指定した場合は，その関数を使用してマップの順序が判別されます．

一般に，キーとして使用するすべてのオブジェクトには，デフォルトコンストラクタを定義し，必要な比較演算子をオーバーロードする必要があります．

mapクラスには次の比較演算子が定義されています．

**map クラスの比較演算子**

**==, <, <=, !=, >, >=**

map クラスで定義されているメンバ関数を表 14-8 に示します．説明の中の key_type はキーの型，value_type は pair <Key, T> を表します．

**表 14-8　map クラスのメンバ関数**

| メンバ関数 | 説明 |
|---|---|
| iterator **begin**( );<br>const_iterator **begin**( ) const; | マップ内の最初の要素を指す反復子を返す |
| void **clear**( ); | マップの要素をすべて削除する |
| size_type **count**(const key_type &*k*) const; | マップ内で *k* が発生する回数(1 または 0)を返す |
| bool **empty**( ) const; | 呼び出し元マップが空である場合は真，空ではない場合は偽を返す |
| iterator **end**( );<br>const_iterator **end**( ) const; | マップの末尾を指す反復子を返す |
| pair<iterator, iterator><br>　**equal_range**(const key_type &*k*);<br>pair<const_iterator, const_iterator><br>　**equal_range**(const key_type &*k*) const; | 指定のキーを含むマップの先頭の要素と末尾の要素を指す 1 組の反復子を返す |
| void **erase**(iterator *i* ); | *i* が指す要素を削除する |
| void **erase**(iterator *start*, iterator *end* ); | *start* から *end* までの範囲の要素を削除する |
| size_type **erase**(const key_type &*k*) | 値 *k* を持つキーの要素を削除する |
| iterator **find**(const key_type &*k*);<br>const_iterator **find**(const key_type &*k*)<br>　const; | 指定したキーを指す反復子を返す．キーが見つからない場合は，マップの末尾を指す反復子を返す |
| allocator_type **get_allocator**( ) const; | マップのアロケータを返す |
| iterator **insert**(iterator *i*,<br>　　　　　const value_type &*val* ); | *i* に指定した位置，またはその後に *val* を挿入する．その要素を指す反復子を返す |
| template <class InIter><br>　void **insert**(InIter *start*, InIter *end* ) | 指定の範囲の要素を挿入する |
| pair<iterator, bool><br>　**insert**(const value_type &*val* ); | 呼び出し元マップに *val* を挿入する．その要素への反復子を返す．要素はすでに存在しない場合にだけ挿入される．要素が挿入された場合は pair<iterator,true> が返される．要素が挿入されなかった場合は pair<iterator,false> が返される |
| key_compare key_comp( ) const; | キーを比較する関数オブジェクトを返す |

（続く）

（続き）

| メンバ関数 | 説明 |
|---|---|
| iterator **lower_bound**(const key_type &*k*);<br>const_iterator<br>　**lower_bound**(const key_type &*k*) const; | *k* 以上のキーを持つマップ内の最初の要素を指す反復子を返す |
| size_type **max_size**( ) const; | マップに格納できる要素の最大数を返す |
| reference **operator[ ]**(const key_type &*i* ) | *i* に指定した要素への参照を返す．この要素が存在しない場合は，この要素が挿入される |
| reverse_iterator **rbegin**( );<br>const_reverse_iterator **rbegin**( ) const; | マップの末尾を指す逆反復子を返す |
| reverse_iterator **rend**( );<br>const_reverse_iterator **rend**( ) const; | マップの先頭を指す逆反復子を返す |
| size_type **size**( ) const; | マップに現在保存されている要素の数を返す |
| void **swap**(map<Key, T, Comp,<br>　　　　Allocator> &*ob*) | 呼び出し元マップに含まれている要素を *ob* の要素と交換する |
| iterator **upper_bound**(const key_type &*k*);<br>const_iterator<br>　**upper_bound**(const key_type &*k*) const; | *k* より大きいキーを持つマップ内の最初の要素を指す反復子を返す |
| value_compare **value_comp**( ) const; | 値を比較する関数オブジェクトを返す |

キー / 値の組み合わせは，pair 型オブジェクトとしてマップに保存されます．pair 型オブジェクトのテンプレート指定を次に示します．

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type;   // キーの型
  typedef Vtype second_type; // 値の型
  Ktype first;   // キーを含む
  Vtype second; // 値を含む

  // コンストラクタ
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

コメントに示したとおり，first にはキー，second にはそのキーに関連付ける値が格納されます．

pair を作成するには，pair のいずれかのコンストラクタを使うか，make_pair() 関数を使用して，仮引数として使用するデータの型に応じて pair オブジェクトを作成します．make_pair() 関数は汎用関数で，プロトタイプは次のとおりです．

make_pair()関数

```
template <class Ktype, class Vtype>
  pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

ご覧のとおり，make_pair()関数はKtypeとVtypeに指定した型の値から成るpairオブジェクトを返します．make_pair()関数の利点は，保存するオブジェクトの型を明示的に指定しなくても，コンパイラが自動的に判別できるという点です．

## 例 14.5　マップ

1. 次のプログラムは，マップの基本的な使用方法を示しています．このプログラムでは10個のキー/値の組み合わせを保存します．キーは文字，値は整数です．次の要領でキー/値の組み合わせを保存します．

```
A       0
B       1
C       2
```

キー/値を保存したら，ユーザーにキーを入力するように求め(AからJまでの文字)，そのキーに関連付けられている値を表示します．

```
// 単純なマップの使用例
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // マップにキー/値を格納する
  for(i=0; i<10; i++) {
    m.insert(pair<char, int>('A'+i, i));
  }

  char ch;
  cout << "Enter key: ";
  cin >> ch;

  map<char, int>::iterator p;
```

```
  // 指定のキーの値を探す
  p = m.find(ch);
  if(p != m.end())
    cout << p->second;
  else
    cout << "Key not in map.¥n";

  return 0;
}
```

このプログラムでは，pairテンプレートクラスを使用してキー/値の組み合わせを作成しています．pairによって指定するデータ型は，pairを挿入するマップのデータ型と同じでなければなりません．

キーと値を使用してマップを初期化したら，find()関数を使用して，特定のキーに対応する値を検索することができます．find()関数は，対応する要素(キーが見つからない場合はマップの末尾)を指す反復子を返します．一致するキーが見つかった場合は，そのキーに対応する値がpairのsecondメンバに格納されます．

**2.** 例1のプログラムでは，pair<char, int>を使用してキー/値の組み合わせを明示的に作成しました．この方法も誤りではありませんが，ほとんどの場合はmake_pair()関数を使用する方が簡単です．この関数を使用すると，仮引数として使用したデータの型を基にpairオブジェクトを作成することができます．たとえば，前述のプログラムの場合は，次のプログラムコードを使用してもキー/値の組み合わせをmに挿入することができます．

```
m.insert(make_pair((char)('A'+i), i));
```

この場合，iとAを加算する際のint型への自動変換を上書きするために，charへの型変換が必要となります．そうしないと，型の判別は自動的に行われます．

**3.** すべてのコンテナと同様，マップには自分で作成した型のオブジェクトを格納することができます．たとえば，次のプログラムでは単語とその反対語のマップを作成しています．このために，wordとoppositeという2つのクラスを作成します．マップではキーがソートされた形で管理されるので，プログラムではword型オブジェクト用に<演算子も定義します．一般に，キーとして使用するクラスに対しては必ず<演算子を定義します(コンパイラによっては，このほかの演算子も定義しなければならないことがあります)．

```
// 反対語のマップ
#include <iostream>
```

```
#include <map>
#include <cstring>
using namespace std;

class word {
  char str[20];
public:
  word() { strcpy(str, ""); }
  word(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

// wordオブジェクトに対する<演算子を定義しなければならない
bool operator<(word a, word b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class opposite {
  char str[20];
public:
  opposite() { strcpy(str, ""); }
  opposite(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

int main()
{
  map<word, opposite> m;

  // 単語と反対語をマップに格納する
  m.insert(pair<word,
    opposite>(word("yes"), opposite("no")));
  m.insert(pair<word,
    opposite>(word("good"), opposite("bad")));
  m.insert(pair<word,
    opposite>(word("left"), opposite("right")));
  m.insert(pair<word,
    opposite>(word("up"), opposite("down")));

  // 単語に対する反対語を探す
  char str[80];
  cout << "Enter word: ";
  cin >> str;

  map<word, opposite>::iterator p;

  p = m.find(word(str));
  if(p != m.end())
```

```
      cout << "Opposite: " <<  p->second.get();
    else
      cout << "Word not in map.¥n";

    return 0;
  }
```

この例では，マップ内の各エントリはヌル終端文字列を保持する文字配列です．この章では後から，標準のstring型を使用してこのプログラムをより簡単に作成する方法を示します．

**練習問題　14.5　マップ**

1. 例に小さな変更を加え，実験しなさい．
2. 名前と電話番号を含むマップを作成しなさい．名前と電話番号を入力することができ，名前を入力して電話番号を検索できるようにしなさい(例3をモデルとして使います)．
3. マップ内でキーとして使用するオブジェクトに対し，<演算子を定義する必要があるかどうか答えなさい．

# 14.6　アルゴリズム

前述のとおり，アルゴリズムはコンテナを処理します．各コンテナは独自の基本処理をサポートしていますが，標準アルゴリズムはさらに高度で複雑な処理をサポートしています．また，標準アルゴリズムを使用すると，さまざまな型のコンテナを同時に処理することができます．STLアルゴリズムにアクセスするためには，プログラムに<algorithm>ヘッダをインクルードします．

STLで定義されている数多くのアルゴリズムを表14-9に示します．すべてのアルゴリズムはテンプレート関数です．つまり，すべての型のコンテナに対して使うことができます．後出の例では，代表的なアルゴリズムの使用方法を示します．

**表14-9　STLアルゴリズム**

| アルゴリズム | 説明 |
|---|---|
| adjacent_find | シーケンス内の近接する一致要素を検索し，一致する最初の要素を指す反復子を返す |
| binary_search | 順序付きシーケンス上で二分探索を行う |
| copy | シーケンスをコピーする |

| アルゴリズム | 説明 |
|---|---|
| copy_backward | copy()と同じ．ただし，シーケンスの最後の要素を前に移動する |
| count | シーケンス内の要素数を返す |
| count_if | シーケンス内で，いくつかの条件式を満たす要素数を返す |
| equal | 2つの範囲が同じかどうかを判別する |
| equal_range | シーケンスの順序を乱すことなく要素をシーケンスに挿入できる範囲を返す |
| fill<br>fill_*n* | 指定の値を使用して範囲内を埋める |
| find | 値の範囲を検索し，最初の要素を指す反復子を返す |
| find_end | 部分シーケンスの範囲を検索する．その範囲内の部分シーケンスの最後の要素を指す反復子を返す |
| find_first_of | シーケンス内で範囲内の要素に一致する最初の要素を検索する |
| find_if | ユーザー定義単項条件式が真を返すような要素範囲を検索する |
| for_each | 範囲内の要素に関数を適用する |
| generate<br>generate_*n* | ジェネレータ関数によって返される値を範囲内の要素に挿入する |
| includes | 1つのシーケンスに別のシーケンス内のすべての要素が含まれるかどうかを判別する |
| inplace_merge | 範囲をほかの範囲と結合する．どちらの範囲も昇順にソートしなければならない．結果のシーケンスはソートされる |
| iter_swap | 2つの反復子引数が指す値を交換する |
| lexicographical_compare | 2つのシーケンスをアルファベット的に比較する |
| lower_bound | シーケンス内で指定の値以上である最初の点を検索する |
| make_heap | シーケンスを基にヒープを作成する |
| max | 2つの値のうち大きい方を返す |
| max_element | 範囲内の最大の要素を指す反復子を返す |
| merge | 2つの順序付きシーケンスを結合し，結果を第3のシーケンスに格納する |
| min | 2つの値のうち小さい方を返す |
| min_element | 範囲内の最小の要素を指す反復子を返す |
| mismatch | 2つのシーケンスの要素間で一致しない最初の要素を検索する．2つの要素を指す反復子を返す |
| next_permutation | シーケンスの次の置換を作成する |
| nth_element | 指定の要素Eより小さいすべての要素がその要素より前になり，Eより大きいすべての要素が後になるように，シーケンスを並べ替える |
| partial_sort | 範囲をソートする |
| partial_sort_copy | 範囲をソートし，結果のシーケンスに収まるだけの要素をコピーする |
| partition | 条件式が真を返すすべての要素が偽を返す要素よりも前になるように，シーケンスを並べ替える |
| pop_heap | 先頭と末尾の要素を交換し，ヒープを作成し直す |

（続く）

（続き）

| アルゴリズム | 説明 |
|---|---|
| prev_permutation | シーケンスの前の置換を作成する |
| push_heap | 要素をヒープの最後に格納する |
| random_shuffle | シーケンスをランダムにする |
| remove<br>remove_if<br>remove_copy<br>remove_copy_if | 指定の要素から範囲を削除する |
| replace<br>replace_copy<br>replace_if<br>replace_copy_if | 範囲内で要素を入れ替える |
| reverse<br>reverse_copy | 範囲要素を逆順にする |
| rotate<br>rotate_copy | 範囲内の要素を左方向に回転する |
| search<br>search_*n* | シーケンス内の部分シーケンスを検索する<br>指定の数の似た要素を持つシーケンスを検索する |
| set_difference | 2 つの順序付きセット内の違いを含むシーケンスを生成する |
| set_intersection | 2 つの順序付きセット内の交差を含むシーケンスを生成する |
| set_symmetric_difference | 2 つの順序付きセット内の対称的な違いを含むシーケンスを生成する |
| set_union | 2 つの順序付きセット内の共用体を含むシーケンスを生成する |
| sort | 範囲をソートする |
| sort_heap | 指定の範囲内のヒープをソートする |
| stable_partition | 条件式が真を返すすべての要素が偽を返す要素よりも前になるように，シーケンスを並べ替える．区切りは変わらない．シーケンスの相対順序は維持される |
| stable_sort | 範囲をソートする．ソートは維持的に行われる．同等な要素が並べ替えられることはない |
| swap | 2 つの要素を交換する |
| swap_ranges | 範囲内の要素を交換する |
| transform | 要素の範囲に関数を適用し，結果を新しいシーケンスに格納する |
| unique<br>unique_copy | 範囲内から重複する要素を除外する |
| upper_bound | シーケンス内で，指定の値以下である最後の点を検索する |

## 例 14.6　アルゴリズム

**1.** アルゴリズムの中で特に単純なのは，count()とcount_if()の2つです．これらのアルゴリズムの一般形式を次に示します．

count()アルゴリズムとcount_if()アルゴリズム

```
template <class InIter, class T>
     size_t count(InIter start, InIter end, const T &val);
template <class InIter, class T>
size_tcount_if(InIterstart,InIterend,UnPredpfn);
```

count()アルゴリズムは，シーケンス内のstartからendまでの範囲で，valに一致する要素の数を返します．count_if()アルゴリズムは，シーケンス内のstartからendまでの範囲で，単項条件式pfnが真を返す要素の数を返します．

次のプログラムは，count()とcount_if()の使用例を示しています．

```
// countとcount_ifの使用例
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* 値が偶数かどうかを判別する
   単項条件式 */
bool even(int x)
{
  return !(x%2);
}

int main()
{
  vector<int> v;
  int i;

  for(i=0; i<20; i++) {
    if(i%2) v.push_back(1);
    else v.push_back(2);
  }

  cout << "Sequence: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;
```

```
  int n;
  n = count(v.begin(), v.end(), 1);
  cout << n << " elements are 1¥n";

  n = count_if(v.begin(), v.end(), even);
  cout << n << " elements are even.¥n";

  return 0;
}
```

このプログラムからの出力は，次のとおりです．

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 elements are 1
10 elements are even.
```

このプログラムでは，まず20個の要素を持つベクトルを作成し，1と2を交互に格納しています．次に，count()アルゴリズムを使用して1の数を調べます．次に，count_if()アルゴリズムを使用して，偶数である要素の数を調べます．単項条件式even()の記述方法に注目してください．すべての単項条件式は，処理対象のコンテナに保存されるデータと同じ型のオブジェクトを仮引数として受け取ります．そして，このオブジェクトに対する結果として真または偽を返さなければなりません．

**2.** 元のシーケンスから取り出した，特定の項目だけを含む新しいシーケンスを作成すると役立つことがあります．この処理を行うアルゴリズムがremove_copy()です．このアルゴリズムの一般形式は，次のとおりです．

remove_copy()アルゴリズム

```
template <class InIter, class OutIter, class T>
 OutIter remove_copy(InIter start, InIter end,
                     OutIter result, const T &val);
```

remove_copy()アルゴリズムは，指定の範囲からvalに一致する要素を削除し，resultが指すシーケンスに結果を格納します．このアルゴリズムは，resultの末尾を指す反復子を返します．出力コンテナには，結果を格納できるだけのサイズが必要です．

次のプログラムは，remove_copy()の使用例を示しています．このプログラムでは1と2を含むシーケンスを作成し，このシーケンスから1をすべて削除しています．

```
// remove_copyの使用例
#include <iostream>
#include <vector>
```

```
#include <algorithm>
using namespace std;

int main()
{
  vector<int> v, v2(20);
  int i;

  for(i=0; i<20; i++) {
    if(i%2) v.push_back(1);
    else v.push_back(2);
  }

  cout << "Sequence: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  // 1を削除する
  remove_copy(v.begin(), v.end(), v2.begin(), 1);
  cout << "Result: ";
  for(i=0; i<v2.size(); i++) cout << v2[i] << " ";
  cout << endl << endl;

  return 0;
}
```

このプログラムからの出力を次に示します.

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Result: 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

**3.** シーケンスを逆順にするreverse()アルゴリズムもよく使用します. このアルゴリズムの一般形式は, 次のとおりです.

reverse()アルゴリズム

```
template <class BiIter> void reverse(BiIter start,
➥BiIter end);
```

reverse()アルゴリズムは*start*と*end*によって指定された範囲を逆順にします.

次のプログラムは, reverse()の使用例を示しています.

```
// reverseの使用例
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int main()
{
  vector<int> v;
  int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Initial: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Reversed: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reversed: 9 8 7 6 5 4 3 2 1 0
```

**4.** 特に興味深いアルゴリズムとして，transform()があります．このアルゴリズムは，指定した関数に従って，範囲内の各要素を修正します．transform()アルゴリズムには次の２つの一般形式があります．

transform()アルゴリズム

```
template <class InIter, class OutIter, class Func)
  OutIter transform(InIter start, InIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter,
class Func)
   OutIter transform(InIter1 start1, InIter1 end1,
                     InIter2 start2,
                     OutIter result, Func binaryfunc);
```

transform()アルゴリズムは，範囲内の要素に関数を適用し，結果を *result* に格納します．1つ目の形式では，*start* と *end* に範囲を指定します．*unaryfunc* には，適用する関数を指定します．この関数は，要素の値を仮引数として受け取り，変形後の要素を返さなければなりません．２つ目の形式では，２項演算子関数を使用して変形を適

用します．2項演算子関数は，変形するシーケンスからの要素値を第1仮引数として受け取り，2つ目のシーケンスからの要素を第2仮引数として受け取ります．どちらの形式も，結果のシーケンスの最後を指す反復子を返します．

次のプログラムでは，xform()という名前の単純な変形関数を使用して，リストの内容を2乗しています．結果のシーケンスを，元のシーケンスを格納していたのと同じリストに格納している点に注目してください．

```
// 変形アルゴリズムの例
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// 単純な変形関数
int xform(int i) {
  return i * i; // 元の値を2乗する
}

int main()
{
  list<int> xl;
  int i;

  // 値をリストに格納する
  for(i=0; i<10; i++) xl.push_back(i);

  cout << "Original contents of xl: ";
  list<int>::iterator p = xl.begin();
  while(p != xl.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // xlを変形する
  p = transform(xl.begin(), xl.end(), xl.begin(), xform);

  cout << "Transformed contents of xl: ";
  p = xl.begin();
  while(p != xl.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

このプログラムからの出力は，次のとおりです．

```
Original contents of xl: 0 1 2 3 4 5 6 7 8 9
Transformed contents of xl: 0 1 4 9 16 25 36 49 64 81
```

ご覧のとおり，xl リスト内の各要素が2乗されています．

## 練習問題 14.6 アルゴリズム

**1.** sort() アルゴリズムの形式は，次のとおりです．

sort()アルゴリズム

```
template <class RandIter>
  void sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void sort(RandIter start, RandIter end, Comp cmpfn);
```

sort() アルゴリズムを使用すると，*start* と *end* によって指定した範囲をソートすることができます．2つ目の形式では，2つの要素の大小を判別するための比較関数を指定することができます．sort() アルゴリズムを使用するプログラムを作成しなさい(任意の形式を使用しなさい)．

**2.** merge() アルゴリズムを使用すると，2つの順序付きシーケンスを結合し，結果を第3のシーケンスに格納することができます．merge() アルゴリズムの一般形式の1つを次に示します．

merge()アルゴリズム

```
template <class InIter1, class InIter2, class
OutIter>
  OutIter merge(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2,
                OutIter result);
```

*start1* と *end1*，*start2* と *end2* には，結合するシーケンスを指定します．結果のシーケンスは *result* が指すシーケンスに格納されます．結果のシーケンスの末尾を指す反復子が返されます．このアルゴリズムを使用するプログラムを作成しなさい．

# 14.7　string クラス

ご存じのとおり，C++では組み込みの文字列型がサポートされていませんが，2種類の方法で文字列を処理することができます．第1の方法は，すでにお馴染みの従来のヌル終端文字配列を使用する方法です．この文字列のことを**C文字列**(C string)と呼ぶことがあります．第2の方法は，string型のクラスオブジェクトを使用する方法です．ここでは，この方法について説明します．

**string** クラスは，実際にはbasic_stringという名前のより汎用的なテンプレートクラスを特化したものです．basic_stringを特化したクラスは，実際には2種類あります．1つは8ビット文字列をサポートするstringクラス，もう1つはワイド文字をサポートするwstringクラスです．一般的なプログラムでは8ビット文字の方がよく使われるので，ここではstringクラスについて説明します．

stringクラスを見る前に，このクラスがなぜC++ライブラリに含まれているのかを理解しておかなければなりません．標準クラスがC++に無計画に追加されることはありません．新しい機能を追加するたびに熟考され，討議が行われています．C++にはすでにヌル終端文字配列という形で文字列サポートが組み込まれていることを考えると，stringクラスを追加するのは一見無用な例外を作ることのように思えます．しかし，実際にはそれは誤りです．というのは，ヌル終端文字列はどの標準C++演算子からも操作できず，またC++式の一部として使用することもできないからです．たとえば，次のプログラムコードを見てみましょう．

```
char s1[80], s2[80], s3[80];

s1 = "one";    // 不可能
s2 = "two";    // 不可能
s3 = s1 + s2; // エラー．許可されていない
```

コメントに示したとおり，C++では代入演算子を使用して文字配列に新しい値を格納することができません(初期化中は例外です)．また，+演算子を使用して2つの文字列を連結することもできません．これらの操作は，次に示すようにライブラリ関数を使って行わなければなりません．

```
strcpy(s1, "one");
strcpy(s2, "two");
strcpy(s3, s1);
strcat(s3, s2);
```

技術的に言うと，ヌル終端文字配列そのものはデータ型ではないので，C++演算子を適用することができません．このため，ごく基本的な文字列操作のプログラムコードも不恰好になります．標準C++演算子を使ってヌル終端文字列を操作できないことが，標準stringクラスを開発する一番の理由となったのです．C++でクラスを定義するときは，C++環境に完全に統合できる新しいデータ型を定義することになることを覚えておいてください．当然ながら，新しいクラスに対して演算子をオーバーロードす

ることができます．したがって，標準stringクラスの追加により，ほかのデータ型を管理するのと同じ方法で，つまり演算子を使用して文字列を管理することができるようになりました．

しかし，標準stringクラスが追加されたもう1つの理由として，安全性があります．経験の浅いプログラマや不注意なプログラマであれば，ヌル終端文字列を格納する配列の境界を越えてしまうことは十分に考えられます．たとえば，標準文字列コピー関数であるstrcpy()について考えてみましょう．この関数には，コピー先の配列の境界をチェックする機能が用意されていません．コピー元配列の文字がコピー先配列の容量よりも多いと，プログラムエラーが発生したり，システムがクラッシュしたりする可能性があります．しかし，標準stringクラスを使えば，このようなエラーを回避できます．

標準stringクラスが追加された理由は，3つにまとめることができます．それは，一貫性(文字列によってデータ型が定義されるようになる)，利便性(標準C++演算子を使用できる)，そして安全性(配列境界を越える心配がない)です．ただし，通常のヌル終端文字列を使うのを完全にやめる必要はまったくないことを忘れないでください．ヌル終端文字列は，依然として文字列を実装する最も効率的な方法です．しかし，処理速度が最優先する必要がない場合には，新しいstringクラスを使用することによって，安全かつ完全に統合された方法で文字列を管理することができるのです．

従来はSTLの一部と考えられていませんでしたが，stringクラスはC++で定義されているコンテナクラスの1つです．つまり，stringクラスは前の節で説明したアルゴリズムをサポートしています．しかし，stringクラスはこれ以外の機能も持っています．stringクラスにアクセスするには，プログラムに<string>ヘッダをインクルードする必要があります．

stringクラスは非常に大きなクラスであり，多くのコンストラクタとメンバ関数を持っています．また，多くのメンバ関数は複数のオーバーロード形式を持っています．このため，この章でstringクラスの内容をすべて説明することはできません．この章では，stringクラスの最もよく使われる機能をいくつか紹介します．stringクラスのしくみについて概要を理解すれば，残りの機能についても自分で簡単に学習できるようになります．

stringクラスには，コンストラクタがいくつかあります．次に，最も一般的に使われる3つのコンストラクタのプロトタイプを示します．

**stringクラス**

```
string( );
string(const char *str);
string(const string &str);
```

1つ目の形式では，空のstringオブジェクトを作成します．2つ目の形式では，*str*が指すヌル終端文字列を使用して，stringオブジェクトを作成します．この形式を使用すると，ヌル終端文字列をstring

オブジェクトに変換することができます．3つ目の形式では，ほかのstringオブジェクトを基にstringオブジェクトを作成します．

stringオブジェクトには，文字列に適用できる演算子が数多く用意されています．次の表に，その一部を示します．

**表 14-10　文字列に適用できる演算子**

| 演算子 | 説明 |
|---|---|
| = | 代入 |
| ++ | 連結 |
| += | 連結代入 |
| == | 等しい |
| != | 等しくない |
| < | より小 |
| <= | 以下 |
| > | より大 |
| >= | 以上 |
| [ ] | 添え字 |
| << | 出力 |
| >> | 入力 |

これらの演算子を使用すると，stringオブジェクトを通常の式の中で使用でき，strcpy()やstrcat()といった関数を呼び出す必要がなくなります．一般には，式の中でstringオブジェクトを通常のヌル終端文字列と組み合わせて使うことができます．たとえば，stringオブジェクトにヌル終端文字列を代入することができます．

+演算子を使用すると，stringオブジェクトをほかのstringオブジェクトを連結したり，stringオブジェクトをC形式の文字列を連結したりすることができます．つまり，次のような組み合わせで連結を行うことができます．

**連結**

```
stringオブジェクト + stringオブジェクト
stringオブジェクト + C文字列
C文字列 + stringオブジェクト
```

また，+演算子を使って，文字列の最後に1文字を連結することもできます．

stringクラスには定数nposが定義されており，これは通常は -1 です．この定数は，作成できる最長文字列の長さを表します．

単純な文字列操作の大部分は文字列演算子を使用して行うことができますが，より複雑な操作や微妙な操作はstringクラスのメンバ関数を使用して実行します．stringクラスのメンバ関数は数多くあり，この章では紹介しきれませんので，ここでは特に一般的なものを紹介します．文字列をほかの文字列に代入するには，assign()関数を使用します．この関数には次の2つの形式があります．

assign()関数

```
string &assign(const string &strob, size_type start,
               size_type num);
string &assign(const char *str, size_type num);
```

1つ目の形式では，strobのうち，*start*に指定した索引位置から始まる*num*個の文字が呼び出し元オブジェクトに代入されます．2つ目の形式では，ヌル終端文字列*str*の最初の*num*個の文字が呼び出し元オブジェクトに代入されます．どちらの場合も，呼び出し元オブジェクトへの参照が返されます．言うまでもありませんが，1つの文字列全体を代入する際には，=演算子を使う方がはるかに簡単です．assign()関数が必要となるのは，部分文字列を代入する場合だけです．

append()メンバ関数を使うと，文字列の一部をほかの文字列に追加することができます．この関数には次の2つの形式があります．

append()関数

```
string &append(const string &strob, size_type start,
               size_type num);
string &append(const char *str, size_type num);
```

1つ目の形式では，strobのうち，*start*に指定した索引位置から始まる*num*個の文字が呼び出し元オブジェクトに追加されます．2つ目の形式では，ヌル終端文字列*str*の最初の*num*個の文字が呼び出し元オブジェクトに追加されます．どちらの場合も，呼び出し元オブジェクトへの参照が返されます．言うまでもありませんが，1つの文字列を別の文字列に追加する際には，+演算子を使う方がはるかに簡単です．append()関数が必要となるのは，部分文字列を追加する場合だけです．

insert()関数を使うと，文字列内に文字を挿入することができます．またreplace()関数を使うと，文字列内の文字を置換することができます．これらの関数の形式で，特によく使われるものを次に示します．

insert()関数と replace()関数

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string
                &strob);
string &replace(size_type start, size_type orgNum, const string
           &strob, size_type replaceStart, size_type replaceNum);
```

1つ目の形式のinsert()関数は，呼び出し元オブジェクトの*start*に指定した索引位置に*strob*を挿入します．2つ目の形式のinsert()関数は，*strob*の*insStart*から始まる*num*個の文字を，呼び出し元オブジェクトの*start*に指定した索引位置に挿入します．

1つ目の形式のreplace()関数は，呼び出し元stringオブジェクトの*start*から始まる*num*個の文字を*strob*と置換します．2つ目の形式では，呼び出し元stringオブジェクトの*start*から始まる*orgNum*文字を，*strob*の*replaceStart*から始まる*replaceNum*個の文字と置換します．どちらの場合も，呼び出し元オブジェクトへの参照が返されます．

erase()関数を使うと，文字列から文字を削除することができます．この関数の形式を次に示します．

erase()関数

```
string &erase(size_type start = 0, size_type num = npos);
```

この関数は，呼び出し元オブジェクトの*start*に指定した位置から始まる*num*個の文字を削除します．呼び出し元stringオブジェクトへの参照が返されます．

stringクラスは，文字列を検索するメンバ関数をいくつか持っていますが，その中にfind()，rfind()があります．次に，これらの2つの関数のプロトタイプのうち，最もよく使われるものを示します．

find()関数と rfind()関数

```
size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;
```

find()関数は，*start*に指定した位置を開始位置として，*strob*に指定した文字列と一致する文字列のうち，最初に出現するものを呼び出し元stringオブジェクト内で検索します．検索文字列が見つかった場合，この関数は，呼び出し元stringオブジェクト内の一致する文字列のインデックスを返します．検索文字列が見つからない場合は，nposを返します．rfind()関数は，find()関数の逆の処理を行います．rfind()関数は，*start*に指定した位置を開始位置として，*strob*に指定した文字列と一致する文字

列のうち，最初に出現するものを呼び出し元stringオブジェクト内で逆方向に検索します(つまり，呼び出し元stringオブジェクト内における*strob*と一致する文字列のうち，最後に出現するものを検索します)．検索文字列が見つかった場合，この関数は呼び出し元stringオブジェクト内の一致する文字列のインデックスを返します．検索文字列が見つからない場合は，nposを返します．

1つの文字列の内容全体をほかの文字列の内容全体と比較するには，前述のオーバーロード関係演算子を使うのが一般的です．しかし，文字列の一部分をほかの文字列の一部分と比較したい場合は，compare()メンバ関数を使う必要があります．この関数の形式を次に示します．

compare()関数

```
int compare(size_type start, size_type num, const string &strob)
const;
```

*strob*内でstartを開始位置とする*num*個の文字が，呼び出し元オブジェクトに対して比較されます．呼び出し元stringオブジェクトが*strob*より小さい場合は，0より小さい値が返されます．呼び出し元stringオブジェクトが*strob*より大きい場合は，0より大きい値が返されます．*strob*が呼び出し元stringオブジェクトと等しい場合は，0が返されます．

stringオブジェクトはそのままで使用しても便利ですが，stringオブジェクトをヌル終端文字配列で表したものを取得しなければならないことがあります．たとえば，stringオブジェクトを使用してファイル名を作成するとします．しかし，ファイルを開く際には，標準ヌル終端文字列を指すポインタを指定しなければなりません．この問題を解決するために，メンバ関数c_str()が用意されています．この関数のプロトタイプを次に示します．

c_str()関数

```
const char *c_str( ) const;
```

この関数は，呼び出し元stringオブジェクトに含まれる文字列をヌル終端文字に変換し，その文字列を指すポインタを返します．ヌル終端文字列を変更してはいけません．また，stringオブジェクトに対してほかの操作を行った後の有効性は保証されません．

stringクラスはコンテナなので，文字列の先頭と末尾を指す反復子を返すbegin()関数とend()関数をサポートしています．また，文字列内に現在含まれる文字の数を返すsize()関数もサポートしています．

## 例 14.7　string クラス

**1.** 従来のC形式の文字列を使う方法も単純ではありますが，C++ の string クラスを使用すると，ごく簡単に文字列を処理することができます．たとえばstringオブジェクトでは，代入演算子を使用して引用符で囲んだ文字列をstringオブジェクトに代入したり，+演算子を使用して文字列を連結したり，比較演算子を使用して2つの文字列を比較したりすることができます．次のプログラムでは，これらの操作を行っています．

```
// 文字列操作の短い例
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Demonstrating Strings");
  string str2("String Two");
  string str3;

  // 文字列を代入する
  str3 = str1;
  cout << str1 << "¥n" << str3 << "¥n";

  // 2つの文字列を連結する
  str3 = str1 + str2;
  cout << str3 << "¥n";

  // 2つの文字列を比較する
  if(str3 > str1) cout << "str3 > str1¥n";
  if(str3 == str1+str2)
    cout << "str3 == str1+str2¥n";

  /* stringオブジェクトに
     通常の文字列を代入することができる */
  str1 = "This is a normal string.¥n";
  cout << str1;

  // ほかのstringオブジェクトを使用してstringオブジェクトを作成する
  string str4(str1);
  cout << str4;

  // 文字列を入力する
  cout << "Enter a string: ";
  cin >> str4;
  cout << str4;
```

```
  return 0;
}
```

このプログラムからの出力を次に示します.

```
Demonstrating Strings
Demonstrating Strings
Demonstrating StringsString Two
str3 > str1
str3 == str1+str2
This is a normal string.
This is a normal string.
Enter a string: Hello
Hello
```

ご覧のとおり，string型のオブジェクトはC++の組み込みデータ型を操作するのと似た方法で操作することができます．実際に，これがstringクラスを使用する主な利点です.

文字列をごく簡単に処理できることに注目してください．たとえば，文字列を連結するには+を使用し，2つの文字列を比較するには>を使用しています．C++形式のヌル終端文字列を使用してこれらの操作を行うには，利便性の低いstrcat()関数やstrcmp()関数を呼び出さなければなりません．C++のstringオブジェクトはC形式のヌル終端文字列と自由に組み合わせることができるので，stringオブジェクトをプログラム内で使用することに欠点はありません．そして大きな利点だけを得ることができます.

このプログラムには，注目すべき点がもう1つあります．それは，文字列のサイズを指定していないとう点です．stringオブジェクトのサイズは，指定の文字列を収めるために自動的に調整されます．したがって，文字列を代入したり連結したりする場合，あて先の文字列は新しい文字列を収容するために必要に応じて大きくなります．文字列の終端を越えることはありません．stringオブジェクトのこの動的な性質は，標準ヌル終端文字列(これは境界を越える可能性があります)よりも優れていると考えられる理由の1つです.

**2.** 次のプログラムでは，insert()関数，erase()関数，replace()関数を使用しています.

```
// insert()関数，erase()関数，replace()関数の使用例
#include <iostream>
#include <string>
using namespace std;

int main()
```

```
{
  string str1("This is a test");
  string str2("ABCDEFG");

  cout << "Initial strings:¥n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "¥n¥n";

  // insert()関数の使用例
  cout << "Insert str2 into str1:¥n";
  str1.insert(5, str2);
  cout << str1 << "¥n¥n";

  // erase()関数の使用例
  cout << "Remove 7 characters from str1:¥n";
  str1.erase(5, 7);
  cout << str1 <<"¥n¥n";

  // replace()関数の使用例
  cout << "Replace 2 characters in str1 with str2:¥n";
  str1.replace(5, 2, str2);
  cout << str1 << endl;

  return 0;
}
```

このプログラムからの出力を次に示します.

```
Initial strings:
str1: This is a test
str2: ABCDEFG

Insert str2 into str1:
This ABCDEFGis a test

Remove 7 characters from str1:
This is a test

Replace 2 characters in str1 with str2:
This ABCDEFG a test
```

**3.** stringはデータ型を定義するので，string型オブジェクトを格納するコンテナを作成することができます．例として，14.5節の例3に示した単語と反対語のプログラムを作成するより優れた方法を示します．

```
// stringオブジェクトを使用して単語と反対語を対応付ける
#include <iostream>
#include <map>
#include <string>
```

```
using namespace std;

int main()
{
  map<string, string> m;
  int i;

  m.insert(pair<string, string>("yes", "no"));
  m.insert(pair<string, string>("up", "down"));
  m.insert(pair<string, string>("left", "right"));
  m.insert(pair<string, string>("good", "bad"));

  string s;
  cout << "Enter word: ";
  cin >> s;

  map<string, string>::iterator p;

  p = m.find(s);
  if(p != m.end())
    cout << "Opposite: " << p->second;
  else
    cout << "Word not in map.¥n";

  return 0;
}
```

## 練習問題 14.7　string クラス

**1.** string 型オブジェクトを使用して，次の文字列をリストに格納しなさい．

```
one     two     three     four     five
six     seven   eight     nine     ten
```

次に，リストをソートしなさい．最後に，ソート済みリストを表示しなさい．

**2.** stringはコンテナなので，標準アルゴリズムを使うことができます．ユーザーから文字列の入力を受け取るプログラムを作成しなさい．そして，count()アルゴリズムを使用して，文字列内のeの数を数えなさい．

**3.** 練習問題2で作成したプログラムを修正し，小文字の数を報告するようにしなさい(count_if()アルゴリズムを使用します)．

**4.** string クラスは，何のテンプレートクラスを特化したものか答えなさい．

## この章の理解度チェック

この段階で，次の問題に答えられるかどうか確認しましょう．

1. STLを使用すると，どのような点で信頼性の高いプログラムを容易に作成できるのかを説明しなさい．

2. STLのコンテナ，反復子，アルゴリズムについて説明しなさい．

3. 10個の要素を含むベクトルを作成し，1から10までを含めるプログラムを作成しなさい．次に，このベクトルからリストに偶数の要素だけをコピーしなさい．

4. stringデータ型を使用する利点と欠点を1つずつ挙げなさい．

5. 条件式とは何か説明しなさい．

6. 14.5節の練習問題2で作成したプログラムを，stringオブジェクトを使用して修正しなさい．

7. STL関数オブジェクトについて学習しなさい．手始めとして，2つの標準クラスであるunary_functionとbinary_functionについて調べなさい．これらは関数オブジェクトの作成に役立ちます．

8. コンパイラに付属しているSTLのマニュアルを使用して学習しなさい．興味深い機能や手法を学ぶことができます．

## 総合理解度チェック

次の問題を解き，この章で学んだ知識を，前の章までに学んだ知識に統合できたかどうかを確認しましょう．

1. 第1章から長い道のりでした．少し時間を取って，本書をざっと再読してください．そして，本書で紹介したサンプルプログラム(特に第6章までで紹介したもの)を，本書で学んだC++の全機能を活用して改良する方法を考えてください．

2. プログラミングを習得する最良の方法は，実際に試してみることです．C++プログラムを数多く作成してください．C++特有の機能の使い方を練習してください．

**3.** STLについてさらに学習してください．アルゴリズムでコンテナを操作することにより，プログラムを縮小できることが多いため，将来的には多くのプログラミング作業がSTLを中心に形成されるようになります．

**4.** C++によって絶大な力を得ることができることを覚えておいてください．この力を賢明な方法で使わなければなりません．この力によって，プログラミングの能力を高めることができます．しかしこの力を誤用すれば，理解しづらく，ほとんど追跡が不可能で，非常に管理しづらいプログラムが出来上がることもあります．C++は強力なツールです．しかし，すべてのツールについて言えるように，その力を発揮できるかどうかは使う人次第なのです．

# 付録 A

## C と C++ の相違点

C++は，その大部分はANSI標準Cのスーパーセットであり，ほとんどすべてのCプログラムはC++プログラムでもあります．ただし両者の間には相違点がいくつかあり，その一部については第1章で紹介しました．ここでは，そのほかに注意すべき相違点を示します．

- Cでは，文字定数は自動的に整数と見なされるのに対し，C++ではそう見なされない．これは小さいながらも重要な相違点である．
- Cでは，グローバル変数を複数回宣言してもエラーとはならない(ただし，これは良いプログラミング習慣ではない)．C++ではエラーとなる．
- Cの識別子は，少なくとも31文字までは有効である．C++では，有効文字数の制限はない．ただし実用面から言うと，長すぎる識別子は扱いにくく，必要となることもほとんどない．
- Cでは，プログラム内からmain()を呼び出すことができる(ただしあまり一般的ではない)．C++では呼び出すことができない．
- Cでは，レジスタ変数のアドレスを取得することができない．C++では取得できる．
- Cではwchar_t型はtypedefとして定義されているが，C++ではwchar_tはキーワードである．

# 付録 B

## 解答

注：紙面の都合上，本来1行で記述すべきコードが2行にわたっている場合があります．「➡」という記号を挿入した箇所がそれに該当します．この記号が入っている行とその前の行は1行で記述してください．

## 1.1：練習問題

**1.** （省略）

## 1.2：練習問題

**1.** （省略）

## 1.3：練習問題

**1.**
```
#include <iostream>
using namespace std;

int main()
{
  double hours, wage;

  cout << "Enter hours worked: ";
  cin >> hours;

  cout << "Enter wage per hour: ";
  cin >> wage;

  cout << "Pay is: $" << wage * hours;

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

int main()
{
  double feet;

  do {
     cout << "Enter feet (0 to quit): ";
     cin >> feet;

     cout << feet * 12 << " inches¥n";
  } while (feet != 0.0);

  return 0;
}
```

**3.**
```
/*
   このプログラムは，最小公約数を計算する
*/
#include <iostream>
using namespace std;

int main()
{
  int a, b, d, min;

  cout << "Enter two numbers: ";
  cin >> a >> b;

  min = a > b ? b : a;

  for(d = 2; d<min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
    cout << "No common denominators¥n";
    return 0;
  }
  cout << "The lowest common denominator is " << d
  ➡<< ".¥n";

  return 0;
}
```

## 1.4：練習問題

**1.** このコメントは，変わってはいますが有効です．

**2.** （省略）

## 1.5：練習問題

**1.** （省略）

**2.**
```
#include <iostream>
#include <cstring>
using namespace std;

class card {
  char title[80];  // 本のタイトル
  char author[40]; // 著者
  int number;      // 冊数
public:
  void store(char *t, char *name, int num);
  void show();
};

void card::store(char *t, char *name, int num)
{
  strcpy(title, t);
  strcpy(author, name);
  number = num;
}

void card::show()
{
  cout << "Title: " << title << "¥n";
  cout << "Author: " << author << "¥n";
  cout << "Number on hand: " << number << "¥n";
}

int main()
{
  card book1, book2, book3;

  book1.store("Dune", "Frank Herbert", 2);
  book2.store("The Foundation Trilogy", "Isaac
  ➡Asimov", 2);
  book3.store("The Rainbow", "D. H. Lawrence", 1);

  book1.show();
  book2.show();
  book3.show();

  return 0;
}
```

**3.**
```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
  int queue[SIZE]; // キューを保存する
  int head, tail;  // 先頭と末尾の索引
public:
  void init();     // 初期化
  void q(int num); // 設定
  int deq();       // 取得
};

// 初期化
void q_type::init()
{
  head = tail = 0;
}

// キューに値を格納する
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full¥n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // 循環する
  queue[tail] = num;
}

// キューから値を削除する
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty¥n";
    return 0;  // またはその他のエラーインジケータ
  }
  head++;
  if(head==SIZE) head = 0; // 循環する
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;
```

```
    q1.init();
    q2.init();

    for(i=1; i<=10; i++) {
      q1.q(i);
      q2.q(i*i);
    }

    for(i=1; i<=10; i++) {
      cout << "Dequeue 1: " << q1.deq() << "¥n";
      cout << "Dequeue 2: " << q2.deq() << "¥n";
    }

    return 0;
  }
```

## 1.6：練習問題

1. f()関数にプロトタイプがありません．
2. （省略）

## 1.7：練習問題

1.

```
#include <iostream>
#include <cmath>
using namespace std;

// 整数，長整数，倍精度浮動少数点数用にsroot()関数をオーバーロードする

int sroot(int i);
long sroot(long i);
double sroot(double i);

int main()
{
  cout << "Square root of 90.34 is : " <<
  ➡sroot(90.34);
  cout << "¥n";
  cout << "Square root of 90L is : " << sroot(90L);
  cout << "¥n";
  cout << "Square root of 90 is : " << sroot(90);

  return 0;
}

// 整数の平方根を返す
int sroot(int i)
{
  cout << "computing integer root¥n";
  return (int) sqrt((double) i);
}

// 長整数の平方根を返す
long sroot(long i)
{
  cout << "computing long root¥n";
  return (long) sqrt((double) i);
}

// 倍精度浮動少数点数の平方根を返す
double sroot(double i)
{
  cout << "computing double root¥n";
  return sqrt(i);
}
```

2. atof()，atoi()，atol()の各関数は，オーバーロードできません．これらの関数の違いは，戻り値のデータ型だけだからです．関数をオーバーロードするためには，引数の型または数が違っていなければなりません．

3.

```
// min()関数をオーバーロードする

#include <iostream>
#include <cctype>
using namespace std;

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

int main()
{
  cout << "Min is: " << min('x', 'a') << "¥n";
  cout << "Min is: " << min(10, 20) << "¥n";
  cout << "Min is: " << min(0.2234, 99.2) << "¥n";

  return 0;
}

// 文字用のmin()
char min(char a, char b)
{
  return tolower(a)<tolower(b) ? a : b;
}

// 整数用のmin()
int min(int a, int b)
{
  return a<b ? a : b;
}

// 倍精度浮動少数点数用のmin()
double min(double a, double b)
{
  return a<b ? a : b;
}
```

4.

```
#include <iostream>
using namespace std;

// sleepをオーバーロードし，整数またはchar *を引数として受け取る
void sleep(int n);
void sleep(char *n);

// プロセッサの速度に合わせてこの値を変更する
#define DELAY 100000

int main()
{
  cout << '.';
  sleep(3);
  cout << '.';
  sleep("2");
  cout << '.';

  return 0;
}

// 整数を受け取るSleep()
void sleep(int n)
{
  long i;

  for( ; n; n--)
    for(i=0; i<DELAY; i++) ;
}

// char *引数を受け取るSleep()
void sleep(char *n)
{
  long i;
  int j;

  j = atoi(n);

  for( ; j; j--)
    for(i=0; i<DELAY; i++) ;
}
```

## 第1章：この章の理解度チェック

1. ポリモーフィズムとは，1つの汎用インターフェイスを使用して，複数の異なる機能にアクセスするためのしくみです．カプセル化は，プログラムコードとそれに関するデータの間のつながりを保護します．カプセル化されたルーチンへのアクセスは厳密に制御されており，これによって不要な干渉を防ぐことができます．継承とは，1つのオブジェクトがほかのオブジェクトの特性を受け継ぐプロセスのことです．継承を使うと，階層的分類システムを実現することができます．

2. C++プログラムにコメントを含めるには，通常のC形式のスタイルを使うか，C++独自の形式の単一行コメントを使用します．

3.

```
#include <iostream>
using namespace std;

int main()
{
  int b, e, r;
  cout << "Enter base: ";
  cin >> b;
  cout << "Enter exponent: ";
  cin >> e;

  r = 1;
  for( ; e; e--) r = r * b;

  cout << "Result: " << r;

  return 0;
}
```

**4.**
```
#include <iostream>
#include <cstring>
using namespace std;

// 文字列取得関数をオーバーロードする
void rev_str(char *s);          // 文字列をその場で逆順にする
void rev_str(char *in, char *out); // 逆順にした文字列を
                                   // outに格納する

int main()
{
  char s1[80], s2[80];
  strcpy(s1, "This is a test");

  rev_str(s1, s2);
  cout << s2 << "¥n";

  rev_str(s1);
  cout << s1 << "¥n";

  return 0;
}

// 文字列を逆順にし, 結果をsに格納する
void rev_str(char *s)
{
  char temp[80];
  int i, j;

  for(i=strlen(s)-1, j=0; i>=0; i--, j++)
    temp[j] = s[i];

  temp[j] = '¥0'; // ヌルで終了する結果

  strcpy(s, temp);
}

// 文字列を逆順にし, 結果をoutに格納する
void rev_str(char *in, char *out)
{
  int i, j;

  for(i=strlen(in)-1, j=0; i>=0; i--, j++)
    out[j] = in[i];

  out[j] = '¥0';  // ヌルで終了する結果
}
```

**5.**
```
#include <iostream.h>

int f(int a);

int main()
{
  cout << f(10);

  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

**6.** boolデータ型にはブール型の値を格納できます．bool型のオブジェクトは，真または偽のどちらかの値しか持ちません．

## 第２章：前章の理解度チェック

**1.**
```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char s[80];

  cout << "Enter a string: ";
  cin >> s;

  cout << "Length: " << strlen(s) << "¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
#include <cstring>
using namespace std;

class addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[10];
public:
  void store(char *n, char *s, char *c, char *t,
             char *z);
  void display();
};
void addr::store(char *n, char *s, char *c, char *t,
                 char *z)
{
  strcpy(name, n);
  strcpy(street, s);
  strcpy(city, c);
  strcpy(state, t);
  strcpy(zip, z);
}

void addr::display()
{
  cout << name << "¥n";
  cout << street << "¥n";
  cout << city << "¥n";
  cout << state << "¥n";
  cout << zip << "¥n¥n";
}

int main()
{
  addr a;

  a.store("C. B. Turkle", "11 Pinetree Lane",
  ➥"Wausau", "In", "46576");

  a.display();

  return 0;
}
```

**3.**
```
#include <iostream>
using namespace std;

int rotate(int i);
long rotate(long i);

int main()
{
  int a;
  long b;

  a = 0x8000;
  b = 8;
  cout  << rotate(a);
  cout << "¥n";
  cout << rotate(b);

  return 0;
}

int rotate(int i)
{
  int x;

  if(i & 0x8000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}

long rotate(long i)
{
  int x;

  if(i & 0x80000000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}
```

**4.** 整数iはmyclassクラスの非公開メンバなので，main()関数からアクセスすることはできません．

## 2.1：練習問題

1.
```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
  int queue[SIZE]; // キューを保持する
  int head, tail;  // 先頭と末尾の索引
public:
  q_type();        // コンストラクタ
  void q(int num); // 設定
  int deq();       // 取得
};

// コンストラクタ
q_type::q_type()
{
  head = tail = 0;
}

// キューに値を格納する
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full¥n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // 循環する
  queue[tail] = num;
}

// キューから値を削除する
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty¥n";
    return 0;  // またはその他のエラーインジケータ
  }
  head++;
  if(head==SIZE) head = 0; // 循環する
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i*i);
  }

  for(i=1; i<=10; i++) {
    cout << "Dequeue 1: " << q1.deq() << "¥n";
    cout << "Dequeue 2: " << q2.deq() << "¥n";
  }

  return 0;
}
```

2.
```
// ストップウォッチエミュレータ
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
  double begin, end;
public:
  stopwatch();
  ~stopwatch();
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "Stopwatch object being destroyed...";
  show();
}

void stopwatch::start()
{
  begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
  end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
  cout << "Elapsed time: " << end - begin;
  cout << "¥n";
}

int main()
{
  stopwatch watch;
  long i;

  watch.start();
  for(i=0; i<320000; i++) ; // forループの時間を計測する
  watch.stop();

  watch.show();

  return 0;
}
```

3. コンストラクタに戻り値型を指定することはできません．

## 2.2：練習問題

1.
```
// 動的にスタックを割り当てる
#include <iostream>
#include <cstdlib>
using namespace std;

// 文字用のstackクラスを宣言する
class stack {
  char *stck;          // スタック領域を確保する
  int tos;             // スタック先頭の索引
  int size;            // スタックのサイズ
public:
  stack(int s);        // コンストラクタ
  ~stack();            // デストラクタ
  void push(char ch); // スタックに文字をプッシュする
  char pop();          // スタックから文字をポップする
};

// スタックを初期化する
stack::stack(int s)
{
  cout << "Constructing a stack¥n";
  tos = 0;
  stck = (char *) malloc(s);
  if(!stck) {
    cout << "Allocation error¥n";
    exit(1);
  }
  size = s;
}

stack::~stack()
{
  free(stck);
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==size) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0;          // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
```

```
  // 自動的に初期化される2つのスタックを作成する
  stack s1(10), s2(10);
  int i;
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop()
  ➥<< "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop()
  ➥<< "¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
#include <ctime>
using namespace std;

class t_and_d {
  time_t systime;
public:
  t_and_d(time_t t); // コンストラクタ
  void show();
};

t_and_d::t_and_d(time_t t)
{
  systime = t;
}

void t_and_d::show()
{
  cout << ctime(&systime);
}

int main()
{
  time_t x;

  x = time(NULL);

  t_and_d ob(x);

  ob.show();

  return 0;
}
```

**3.**
```
#include <iostream>
using namespace std;

class box {
  double l, w, h;
  double volume;
public:
  box(double a, double b, double c);
  void vol();
};

box::box(double a, double b, double c)
{
  l = a;
  w = b;
  h = c;

  volume = l * w * h;
}

void box::vol()
{
  cout << "Volume is: " << volume << "¥n";
}

int main()
{
  box x(2.2, 3.97, 8.09), y(1.0, 2.0, 3.0);

  x.vol();
  y.vol();
  return 0;
}
```

## 2.3：練習問題

**1.**
```
#include <iostream>
using namespace std;

class area_cl {
public:
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w);
  double area();
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w);
  double area();
};

rectangle::rectangle(double h, double w)
{
  height = h;
  width = w;
}

isosceles::isosceles(double h, double w)
{
  height = h;
  width = w;
}

double rectangle::area()
{
  return width * height;
}

double isosceles::area()
{
  return 0.5 * width * height;
}

int main()
{
  rectangle b(10.0, 5.0);
  isosceles i(4.0, 6.0);

  cout << "Rectangle: " << b.area() << "¥n";
  cout << "Triangle: " << i.area() << "¥n";

  return 0;
}
```

## 2.5：練習問題

**1.**
```
// 構造体を使用したstackクラス
#include <iostream>
using namespace std;

#define SIZE 10

// 構造体を使用して，文字用のstackクラスを宣言する
struct stack {
  stack();              // コンストラクタ
  void push(char ch);   // スタックに文字をプッシュする
  char pop();           // スタックから文字をポップする
private:
  char stck[SIZE];      // スタック領域を確保する
  int tos;              // スタック先頭の索引
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
```

```
    return 0;            // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
  // 自動的に初期化される2つのスタックを作成する
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop()
➥ << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop()
➥ << "¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

union swapbytes {
  unsigned char c[2];
  unsigned i;
  swapbytes(unsigned x);
  void swp();
};

swapbytes::swapbytes(unsigned x)
{
  i = x;
}
void swapbytes::swp()
{
  unsigned char temp;

  temp = c[0];
  c[0] = c[1];
  c[1] = temp;
}

int main()
{
  swapbytes ob(1);

  ob.swp();
  cout << ob.i;

  return 0;
}
```

**3.** 無名共用体とは，2つの変数が同じメモリ領域を共有するためのしくみのことです．無名共用体のメンバは，オブジェクトを参照することなく直接アクセスできます．無名共用体のメンバは，共用体自身と同じスコープレベルを持ちます．

## 2.6：練習問題

**1.**
```
#include <iostream>
using namespace std;

// abs()関数を3とおりにオーバーロードする

// 整数用のabs()
inline int abs(int n)
{
  cout << "In integer abs()¥n";
  return n<0 ? -n : n;
}

// 長整数用のabs()
inline long abs(long n)
{
  cout << "In long abs()¥n";
  return n<0 ? -n : n;
}

// 倍精度浮動少数点数用のabs()
inline double abs(double n)
{
  cout << "In double abs()¥n";
  return n<0 ? -n : n;
}

int main()
{
  cout << "Absolute value of -10: " << abs(-10) <<
➥ "¥n";
  cout << "Absolute value of -10L: " << abs(-10L) <<
➥ "¥n";
  cout << "Absolute value of -10.01: " << abs(-
➥ 10.01) << "¥n";

  return 0;
}
```

**2.** この関数にはforループが含まれているため，インライン化できない可能性があります．ループを含む関数は，コンパイラによってはインライン化できません．

## 2.7：練習問題

**1.**
```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字用のstackクラスを宣言する
class stack {
  char stck[SIZE]; // スタック領域を確保する
  int tos;         // スタック先頭の索引
public:
  stack() { tos = 0; }
  void push(char ch)
  {
    if(tos==SIZE) {
      cout << "Stack is full¥n";
      return;
    }
    stck[tos] = ch;
    tos++;
  }
  char pop()
  {
    if(tos==0) {
      cout << "Stack is empty¥n";
      return 0;    // スタックが空の場合はヌルを返す
    }
    tos--;
    return stck[tos];
  }
};

int main()
{
  // 自動的に初期化される2つのスタックを作成する
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop()
➥ << "¥n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop()
➥ << "¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr)
  {
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
      cout << "Allocation error¥n";
      exit(1);
    }
    strcpy(p, ptr);
  }
```

解答 2

```
    ~strtype() { cout << "Freeing p¥n"; free(p); }

    void show()
    {
      cout << p << " - length: " << len;
      cout << "¥n";
    }
};

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  return 0;
}
```

## 第２章：この章の理解度チェック

1. コンストラクタとは，オブジェクトの作成時に呼び出される関数のことです．デストラクタとは，オブジェクトを破棄するときに呼び出される関数のことです．

2.
```
#include <iostream>
using namespace std;

class line {
  int len;
public:
  line(int l);
};

line::line(int l)
{
  len = l;

  int i;
  for(i=0; i<len; i++) cout  << '*';
}

int main()
{
  line l(10);

  return 0;
}
```

3. 10 1000000 -0.0009

4.
```
#include <iostream>
using namespace std;

class area_cl {
public:
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w) { height = h; width
  ➥= w; }
  double area() { return height * width; }
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w) { height = h; width
  ➥= w; }
  double area() { return 0.5 * width * height; }};

class cylinder : public area_cl {
public:
  cylinder(double h, double w) { height = h; width =
  ➥w; }
  double area()
  {
    return (2 * 3.1416 * (width/2) * (width/2)) +
            (3.1416 * width * height);
  }
};

int main()
{
  rectangle b(10.0, 5.0);
  isosceles i(4.0, 6.0);
  cylinder c(3.0, 4.0);

  cout << "Rectangle: " << b.area() << "¥n";
  cout << "Triangle: " << i.area() << "¥n";
  cout << "Cylinder: " << c.area() << "¥n";

  return 0;
}
```

5. インライン関数のプログラムコードは，インラインで展開されます．つまり，関数を実際に呼び出すことはありません．これによって，関数の呼び出しと終了に伴うオーバーヘッドを避けることができます．インライン関数の利点は，実行速度が速まることです．インライン関数の欠点は，プログラムのサイズが増える可能性があることです．

6.
```
#include <iostream>
using namespace std;

class myclass {
  int i, j;
public:
  myclass(int x, int y) { i = x; j = y; }
  void show() { cout << i << " " << j; }
};

int main()
{
  myclass count(2, 3);

  count.show();

  return 0;
}
```

7. クラスのメンバは，デフォルトでは非公開です．構造体のメンバは，デフォルトでは公開されます．

8. 有効です．これは無名共用体を定義するプログラムコードです．

## 第２章：総合理解度チェック

1.
```
#include <iostream>
using namespace std;
class prompt {
  int count;
public:
  prompt(char *s) { cout << s; cin >> count; };
  ~prompt();
};

prompt::~prompt() {
  int i, j;

  for(i=0; i<count; i++) {
    cout << '¥a';
    for(j=0; j<32000; j++) ; // 遅延
  }
}

int main()
{
  prompt ob("Enter a number: ");

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class ftoi {
  double feet;
  double inches;
public:
  ftoi(double f);
};

ftoi::ftoi(double f)
{
  feet = f;
  inches = feet * 12;
  cout << feet << " is " << inches << " inches.¥n";
}

int main()
{
  ftoi a(12.0), b(99.0);
  return 0;
}
```

3.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dice {
```

```
  int val;
public:
  void roll();
};

void dice::roll()
{
  val = (rand() % 6) + 1; // 1から6までを生成する
  cout << val << "¥n";
}

int main()
{
  dice one, two;

  one.roll();
  two.roll();
  one.roll();
  two.roll();
  one.roll();
  two.roll();

  return 0;
}
```

## 第３章：前章の理解度チェック

1. コンストラクタの名前はwidgit()，デストラクタの名前は~widgit() です．

2. コンストラクタ関数は，オブジェクトの作成時（オブジェクトが存在するようになるとき）に呼び出されます．デストラクタはオブジェクトを破棄するときに呼び出されます．

3.
```
class Mars : public planet {
// ...
};
```

4. 関数をインラインで展開するには，関数定義の前にinline指定子を指定するか，関数の定義をクラス宣言に含めます．

5. インライン関数は，最初に使用する前に定義しなければなりません．このほかに主な制限としては，ループを含めることができないこと，再帰してはいけないこと，goto文およびswitch文を含めることができないこと，変数staticを使用できないことが挙げられます．

6.
```
sample ob(100, 'X');
```

## 3.1：練習問題

1. cl1とcl2は2つの別個のクラスであり，クラス型の異なるクラスを代入することはできないので，x＝yという代入文は誤りです．

2.
```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
  int queue[SIZE]; // キューを保存する
  int head, tail;  // 先頭と末尾の索引
public:
  q_type();        // コンストラクタ
  void q(int num); // 設定
  int deq();       // 取得
};

// コンストラクタ
q_type::q_type()
{
  head = tail = 0;
}

// 値をキューに格納する
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full¥n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // 循環する
  queue[tail] = num;
}

// キューから値を削除する
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty¥n";
    return 0;             // またはその他のエラーインジケータ
  }
  head++;
  if(head==SIZE) head = 0; // 循環する
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
  }

  // キューをほかのキューに代入する
  q2 = q1;

  // 両方のキューの内容が同じであることを示す
  for(i=1; i<=10; i++)
    cout << "Dequeue 1: " << q1.deq() << "¥n";

  for(i=1; i<=10; i++)
    cout << "Dequeue 2: " << q2.deq() << "¥n";

  return 0;
}
```

3. キューを格納するメモリを動的に割り当てる場合，1つのキューをほかのキューに代入すると，代入文の左辺にあるキューに割り当てられた動的メモリが失われ，オブジェクトを破棄する際に，右辺のキューに割り当てられたメモリが2回解放されます．どちらの状況も容認できないエラーです．

## 3.2：練習問題

1.
```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字のstackクラスを宣言する
class stack {
  char stck[SIZE];    // スタック領域を確保する
  int tos;            // スタック先頭の索引
public:
  stack();            // コンストラクタ
  void push(char ch); // スタックに文字をプッシュする
  char pop();         // スタックから文字をポップする
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0;         // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}
void showstack(stack o);

int main()
{
  stack s1;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  showstack(s1);
```

解答 3

```
  // main()関数内のs1はまだ存在する
  cout << "s1 stack still contains this: ¥n";
  for(i=0; i<3; i++) cout << s1.pop() << "¥n";

  return 0;
}

// スタックの内容を表示する
void showstack(stack o)
{
  char c;

  // この文が終了すると，oスタックは空になる
  while(c=o.pop()) cout << c << "¥n";
  cout << "¥n";
}
```

このプログラムからの出力は次のようになります．

```
Constructing a stack
c
b
a
Stack is empty
s1 stack still contains this:
c
b
a
```

**2.** neg()関数を呼び出す際に使用したoオブジェクトのpが指す整数を格納するメモリは，main()関数内でまだ必要であるにもかかわらず，neg()関数が終了してoオブジェクトのコピーが破棄されるときに解放されます．

## 3.3：練習問題

**1.**

```
#include <iostream>
using namespace std;

class who {
  char name;
public:
  who(char c) {
     name = c;
     cout << "Constructing who #";
     cout << name << "¥n";
  }
  ~who() { cout << "Destructing who #" << name <<
➡"¥n"; }
};

who makewho()
{
  who temp('B');
  return temp;
}

int main()
{
  who ob('A');

  makewho();

  return 0;
}
```

**2.** オブジェクトを返すのが適さない状況もあります．たとえば，オブジェクトの作成時にディスクファイルを開き，破棄時にそのファイルを閉じる場合，関数からそのオブジェクトを返すと，ファイルは一時オブジェクトを破棄するときに閉じられてしまいます．

## 3.4：練習問題

**1.**

```
#include <iostream>
using namespace std;
class pr2;  // 前方宣言

class pr1 {
  int printing;
  // ...
public:
  pr1() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

class pr2 {
  int printing;
  // ...
public:
  pr2() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

// プリンタが使用中であれば真を返す
int inuse(pr1 o1, pr2 o2)
{
  if(o1.printing || o2.printing) return 1;
  else return 0;
}

int main()
{
  pr1 p1;
  pr2 p2;

  if(!inuse(p1, p2)) cout << "Printer idle¥n";

  cout << "Setting p1 to printing...¥n";
  p1.set_print(1);
  if(inuse(p1, p2)) cout << "Now, printer in
➡use.¥n";

  cout << "Turn off p1...¥n";
  p1.set_print(0);
  if(!inuse(p1, p2)) cout << "Printer idle¥n";

  cout << "Turn on p2...¥n";
  p2.set_print(1);
  if(inuse(p1, p2)) cout << "Now, printer in
➡use.¥n";

  return 0;
}
```

## 第３章：この章の理解度チェック

**1.** オブジェクトを別のオブジェクトに代入するには，両方のオブジェクトのクラス型が同じでなければなりません．

**2.** ob1をob2に代入する際に問題となるのは，ob2の初期値pが指すメモリが代入によって上書きされ，失われるということです．
したがって，このメモリを解放することができなくなり，ob1のpが参照するメモリは，破棄時に2回解放されることになります．これによって動的割り当てシステムが破壊される可能性があります．

**3.**

```
int light(planet p)
{
  return p.get_miles() / 186000;
}
```

**4.** できます．

**5.**

```
// スタックにアルファベットを格納する
#include <iostream>
using namespace std;

#define SIZE 27

// 文字のstackクラスを宣言する
class stack {
  char stck[SIZE];     // スタック領域を確保する
  int tos;             // スタック先頭の索引
public:
  stack();             // コンストラクタ
  void push(char ch); // スタックに文字をプッシュする
  char pop();          // スタックから文字をポップする
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
```

```
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0;           // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);
stack loadstack();

int main()
{
  stack s1;

  s1 = loadstack();
  showstack(s1);

  return 0;
}

// スタックの内容を表示する
void showstack(stack o)

{
  char c;

  // この文が終了すると，oスタックは空になる
  while(c=o.pop()) cout << c << "¥n";
  cout << "¥n";
}

// スタックにアルファベットを格納する
stack loadstack()
{
  stack t;
  char c;
  for(c = 'a'; c <= 'z'; c++) t.push(c);
  return t;
}
```

**6.** 関数にオブジェクトを渡したり，関数からオブジェクトを返すときにはそのオブジェクトのコピーが一時的に作成され，関数の終了時に破棄されます．オブジェクトの一時コピーを破棄する際には，デストラクタ関数によって，プログラム内のほかの箇所で必要なものが破壊されてしまう可能性があります．

**7.** フレンド関数とは，メンバ関数ではありませんが，フレンドであるクラスの非公開メンバにアクセスすることが許可されている関数のことです．

## 第３章：総合理解度チェック

**1.**

```
// スタックにアルファベットを格納する
#include <iostream>
#include <cctype>
using namespace std;

#define SIZE 27

// 文字のstackクラスを宣言する
class stack {
  char stck[SIZE];     // スタック領域を確保する
  int tos;             // スタック先頭の索引
public:
  stack();             // コンストラクタ
  void push(char ch);  // スタックに文字をプッシュする
  char pop();          // スタックから文字をポップする
};

// スタックを初期化する
stack::stack()
{
  cout << "Constructing a stack¥n";
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0;           // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);
stack loadstack();
stack loadstack(int upper);

int main()
{
  stack s1, s2, s3;

  s1 = loadstack();
  showstack(s1);

  // 大文字を取得する
  s2 = loadstack(1);
  showstack(s2);

   // 小文字を使用する
  s3 = loadstack(0);
  showstack(s3);

  return 0;
}
// スタックの内容を表示する
void showstack(stack o)
{
  char c;

  // この文が終了すると，oスタックは空になる
  while(c=o.pop()) cout << c << "¥n";
  cout << "¥n";
}

// スタックにアルファベットを格納する
stack loadstack()
{
  stack t;
  char c;

  for(c = 'a'; c<='z'; c++) t.push(c);
  return t;
}

/* スタックにアルファベット文字を格納する
   upperが1の場合は大文字，1以外の場合は
   小文字を格納する */
stack loadstack(int upper)
{
  stack t;
  char c;

  if(upper) c = 'A';
  else c = 'a';

  for(; toupper(c)<='Z'; c++) t.push(c);
  return t;
}
```

**2.**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
  friend char *get_string(strtype *ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
```

```
  cout << "Freeing p¥n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

char *get_string(strtype *ob)
{
  return ob->p;
}

int main()
{
  strtype s1("This is a test.");

  char *s;

  s1.show();
  // 文字列へのポインタを取得する
  s = get_string(&s1);
  cout << "Here is string contained in s1: ";
  cout << s << "¥n";

  return 0;
}
```

**3.** 派生クラスのオブジェクトを同じクラスから派生したほかのオブジェクトに代入すると，基本クラスのデータもコピーされます．このことを確認するプログラムを次に示します．

```
#include <iostream>
using namespace std;

class base {
  int a;
public:
  void load_a(int n) { a = n; }
  int get_a() { return a; }
};

class derived : public base {
  int b;
public:
  void load_b(int n) { b = n; }
  int get_b() { return b; }
};

int main()
{
  derived ob1, ob2;

  ob1.load_a(5);
  ob1.load_b(10);

  // ob1をob2に代入する
  ob2 = ob1;

  cout << "Here is ob1's a and b: ";
  cout << ob1.get_a() << ' ' << ob1.get_b() << "¥n";
  cout << "Here is ob2's a and b: ";
  cout << ob2.get_a() << ' ' << ob2.get_b() << "¥n";

  return 0;
}
```

## 第４章：前章の理解度チェック

**1.** 1つのオブジェクトを同じ型の別のオブジェクトに代入すると，右辺のオブジェクトのすべてのデータメンバ現在値が左辺のオブジェクトの対応するデータメンバに代入されます．

**2.** 1つのオブジェクトを別のオブジェクトに代入すると，それによって左辺のオブジェクト内にある既存の重要なデータが上書きされ，問題が発生することがあります．たとえば，動的メモリや開いているファイルを指すポインタが上書きされ，失われる可能性があります．

**3.** 関数にオブジェクトを渡すと，オブジェクトのコピーが作成されます．ただし，コピーオブジェクトのコンストラクタ関数は呼び出されません．関数の終了時にオブジェクトを破棄する際には，コピーオブジェクトのデストラクタが呼び出されます．

**4.** オブジェクトを仮引数として渡す際，引数に指定したオブジェクトとそのコピーとの区別が失われる状況は何とおりかあります．たとえば，デストラクタによって動的メモリを 解放した場合，引数に指定したオブジェクトのメモリも解放されます．一般に，引数に指定した元のオブジェクトで必要とされるデータがデストラクタ関数によって破壊されると，元のオブジェクトに被害が及びます．

**5.**
```
#include <iostream>
using namespace std;

class summation {
  int num;
  long sum; // 数値の総計
public:
  void set_sum(int n);
  void show_sum() {
    cout << num << " summed is " << sum << "¥n";
  }
};
void summation::set_sum(int n)
{
  int i;

  num = n;

  sum = 0;
  for(i=1; i<=n; i++)
    sum += i;
}

summation make_sum()
{
  int i;
  summation temp;

  cout << "Enter number: ";
  cin >> i;

  temp.set_sum(i);

  return temp;
}

int main()
{
  summation s;

  s = make_sum();

  s.show_sum();

  return 0;
}
```

**6.** 一部のコンパイラでは，インライン関数にループを含めることはできません．

**7.**
```
#include <iostream>
using namespace std;

class myclass {
  int num;
public:
  myclass(int x) { num = x; }
  friend int isneg(myclass ob);
};

int isneg(myclass ob)
{
  return (ob.num < 0) ? 1 : 0;
}

int main()
{
  myclass a(-1), b(2);

  cout << isneg(a) << ' ' << isneg(b);
  cout << "¥n";

  return 0;
}
```

**8.** フレンド関数は複数のクラスのフレンドとすることができます．

### 4.1：練習問題

**1.**
```
#include <iostream>
using namespace std;

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};
```

```
int main()
{
  letters ob[10] = { 'a', 'b', 'c','d', 'e', 'f',
                     'g', 'h', 'i', 'j' };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

class squares {
  int num, sqr;
public:
  squares(int a, int b) { num = a; sqr = b; }
  void show() {cout << num << ' ' << sqr << "¥n"; }
};

int main()
{
  squares ob[10] = {
    squares(1, 1),
    squares(2, 4),
    squares(3, 9),
    squares(4, 16),
    squares(5, 25),
    squares(6, 36),
    squares(7, 49),
    squares(8, 64),
    squares(9, 81),
    squares(10, 100)
  };
  int i;

  for(i=0; i<10; i++) ob[i].show();

  return 0;
}
```

**3.**
```
#include <iostream>
using namespace std;

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

int main()
{
  letters ob[10] = {
    letters('a'),
    letters('b'),
    letters('c'),
    letters('d'),
    letters('e'),
    letters('f'),
    letters('g'),
    letters('h'),
    letters('i'),
    letters('j')
  };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "¥n";

  return 0;
}
```

## 4.2：練習問題

**1.**
```
// 逆順に表示する
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;

  samp *p;

  p = &ob[3]; // 最後の要素のアドレスを取得する

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "¥n";
    p--;        // 前のオブジェクトに進む
  }

  cout << "¥n";

  return 0;
}
```

**2.**
```
/* オブジェクトの2次元配列を作成する
   ポインタを使用してアクセスする */
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;

  samp *p;

  p = (samp *) ob;
   for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    p++;
    cout << p->get_a() << "¥n";
    p++;
  }

  cout << "¥n";

  return 0;
}
```

## 4.3：練習問題

**1.**
```
// thisポインタを使用する
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int n, int m) { this->a = n; this->b = m; }
  int add() { return this->a + this->b; }
  void show();
};

void myclass::show()
{
  int t;

  t = this->add(); // メンバ関数を呼び出す
  cout << t << "¥n";
}

int main()
{
  myclass ob(10, 14);
```

解答 4

```
  ob.show();

  return 0;
}
```

## 4.4：練習問題

1.
```
#include <iostream>
using namespace std;

int main()
{
  float *f;
  long *l;
  char *c;

  f = new float;
  l = new long;
  c = new char;

  if(!f || !l || !c) {
    cout << "Allocation error.";
    return 1;
  }

  *f = 10.102;
  *l = 100000;
  *c = 'A';

  cout << *f << ' ' << *l << ' ' << *c;
  cout << '¥n';

  delete f; delete l; delete c;

  return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class phone {
  char name[40];
  char number[14];
public:
  void store(char *n, char *num);
  void show();
};

void phone::store(char *n, char *num)
{
  strcpy(name, n);
  strcpy(number, num);
}

void phone::show()
{
  cout << name << ": " << number;
  cout << "¥n";
}

int main()
{
  phone *p;

  p = new phone;

  if(!p) {
    cout << "Allocation error.";
    return 1;
  }

  p->store("Isaac Newton", "111 555-2323");

  p->show();

  delete p;

  return 0;
}
```

3. new演算子が失敗すると，ヌルポインタを返すか例外を生成します．どちらの処置が実行されるかは，使用するコンパイラのマニュアルで調べなければなりません．標準C++では，new演算子はデフォルトで例外を生成します．

## 4.5：練習問題

1.
```
char *p;

p = new char [100];
// ...
strcpy(p, "This is a test");
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  double *p;

  p = new double (-123.0987);

  cout << *p << '¥n';

  return 0;
}
```

## 4.6：練習問題

1.
```
#include <iostream>
using namespace std;

void rneg(int &i); // 参照を使用する
void pneg(int *i); // ポインタを使用する

int main()
{
  int i = 10;
  int j = 20;

  rneg(i);
  pneg(&j);

  cout << i << ' ' << j << '¥n';

  return 0;
}

// 参照仮引数を使用する
void rneg(int &i)
{
  i = -i;
}
// ポインタ仮引数を使用する
void pneg(int *i)
{
  *i = - *i;
}
```

2. triple()関数を呼び出すときに，&演算子によってdのアドレスを明示的に取得しています．これは不要であり，誤りです．参照仮引数を使用する場合は，引数に&演算子を付けません．

3. 参照仮引数のアドレスは，関数に自動的に渡されます．アドレスを手作業で取得する必要はありません．参照呼び出しを使用すると，一般には値呼び出しよりも処理速度が速くなります．引数のコピーは作成されません．したがって，コピーオブジェクトのデストラクタが呼び出されることによる副作用の心配はありません．

## 4.7：練習問題

1. 元のプログラムでは，オブジェクトをshow()関数に値によって渡しています．したがってオブジェクトのコピーが作成されます．show()関数が終了すると，コピーオブジェクトが破棄されてデストラクタが呼び出されます．これによってpが解放されますが，pが指すメモリはshow()関数の引数でまだ必要とされます．次のプログラムは，このプログラムを修正し，参照仮引数を使用して，関数の呼び出し時にコピーオブジェクトが作成されないようにしたものです．

```
// 修正したプログラム
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;
```

```
  p = new char [1];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  strcpy(p, s);
}

// 参照仮引数を使用して修正する
void show(strtype &x)
{
  char *s;

  s = x.get();
  cout << s << "¥n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

## 4.8：練習問題

**1.**

```
// 単純な境界付き2次元配列の例
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
  int isize, jsize;
  int *p;
public:
  array(int i, int j);
  int &put(int i, int j);
  int get(int i, int j);
};

array::array(int i, int j)
{
  p = new int [i*j];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  isize = i;
  jsize = j;
}

// 配列にデータを格納する
int &array::put(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "Bounds error!!!¥n";
    exit(1);
  }
  return p[i*jsize + j];  // p[i]への参照を返す
}

// 配列からデータを取得する
int array::get(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "Bounds error!!!¥n";
    exit(1);
  }
  return p[i*jsize +j];   // 文字を返す
}

int main()
{
  array a(2, 3);
  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      a.put(i, j) = i+j;

  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      cout << a.get(i, j) << ' ';

  // 境界外エラーを生成する
  a.put(10, 10);
```

```
  return 0;
}
```

**2.** 正しくありません．関数から返される参照をポインタに代入することはできません．

## 4.9：練習問題

**1.** （省略）

## 第４章：この章の理解度チェック

**1.**

```
#include <iostream>
using namespace std;

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '¥n'; }
};

int main()
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++)
      ob[i][j].show();

  cout << '¥n';

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '¥n'; }
};

int main()
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };
  a_type *p;

  p = (a_type *) ob;

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++) {
      p->show();
      p++;
    }

  cout << '¥n';

  return 0;
}
```

**3.** thisポインタはメンバ関数に自動的に渡されるポインタで，関数を呼び出したオブジェクトを指します．

**4.** new演算子とdelete演算子の一般形式は次のとおりです．

解答 4

```
p-var = new type;
delete p-var ;
```

new演算子を使うときは，型キャストを行う必要はありません．オブジェクトのサイズは自動的に判別されるので，sizeofを使用する必要はありません．また，プログラムに<cstdlib>をインクルードする必要はありません．

**5.** 参照とは，基本的には暗黙的なポインタ定数のことで，実質的にはほかの変数または引数の別名として使うことができます．参照仮引数を使う利点として，引数のコピーが作成されないということが挙げられます．

**6.**
```
#include <iostream>
using namespace std;

void recip(double &d);

int main()
{
  double x = 100.0;

  cout << "x is " << x << '¥n';

  recip(x);

  cout << "Reciprocal is " << x << '¥n';

  return 0;
}

void recip(double &d)
{
  d = 1/d;
}
```

## 第４章：総合理解度チェック

**1.** ポインタを使ってオブジェクトのメンバにアクセスするには，アロー(->)演算子を使います．

**2.**
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}
strtype::~strtype()
{
  cout << "Freeing p¥n";
  delete [] p;
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "¥n";
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  return 0;
}
```

**3.** （省略）

## 第５章：前章の理解度チェック

**1.** 参照とは特殊なポインタのことで，自動的に間接参照され，参照先のオブジェクトと同義で使うことができます．参照には仮引数参照，独立参照，関数から返される参照の3種類があります．

**2.**
```
#include <iostream>
using namespace std;

int main()
{
  float *f;
  int *i;

  f = new float;
  i = new int;

  if(!f || !i) {
    cout << "Allocation error¥n";
    return 1;
  }

  *f = 10.101;
  *i = 100;

  cout << *f << ' ' << *i << '¥n';

  delete f;
  delete i;

  return 0;
}
```

**3.** 初期値を持つnew演算子の一般形式は次のとおりです．

```
p-var = new type (initializer);
```

たとえば，1つの整数を割り当て，値として10を設定するには次のようにします．

```
int *p;

p = new int (10);
```

**4.**
```
#include <iostream>
using namespace std;

class samp {
  int x;
public:
  samp(int n) { x = n; }
  int getx() { return x; }
};

int main()
{
  samp A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int i;

  for(i=0; i<10; i++) cout << A[i].getx() << ' ';

  cout << "¥n";

  return 0;
}
```

**5.** **利点**：参照仮引数を使用すると，関数を呼び出す際に使用したオブジェクトのコピーが作成されません．参照呼び出しは，一般に値呼び出しよりも高速です．参照仮引数を使うと参照による呼び出しの構文と手続きが効率化されるので，エラーが起こる可能性が減ります．

**欠点**：参照仮引数に変更を加えると，呼び出し時に使用した変数に変更が及びます．参照仮引数を使うと，呼び出し元ルーチンに副作用が及ぶ可能性が生じます．

**6.** できません．

**7.**
```
#include <iostream>
using namespace std;

void mag(long &num, long order);

int main()
{
  long n = 4;
  long o = 2;

  cout << "4 raised to the 2nd order of magnitude is
➥ ";
  mag(n, o);
  cout << n << '¥n';
```

```
  return 0;
}

void mag(long &num, long order)
{
  for( ; order; order--) num = num * 10;
}
```

## 5.1：練習問題

**1.**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype();
  strtype(char *s, int l);
  char *getstring() { return p; }
  int getlength() { return len; }
};

strtype::strtype()
{
  p = new char [255];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  *p = '¥0'; // ヌル文字列
  len = 255;
}

strtype::strtype(char *s, int l)
{
  if(strlen(s) >= l) {
    cout << "Allocating too little memory!¥n";
    exit(1);
  }

  p = new char [l];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, s);
  len = l;
}

int main()
{
  strtype s1;
  strtype s2("This is a test", 100);

  cout << "s1: " << s1.getstring() << " - Length: ";
  cout << s1.getlength() << '¥n';

  cout << "s2: " << s2.getstring() << " - Length: ";
  cout << s2.getlength() << '¥n';

  return 0;
}
```

**2.**

```
// ストップウォッチエミュレータ
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
  double begin, end;
public:
  stopwatch();
  stopwatch(clock_t t);
  ~stopwatch();
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::stopwatch(clock_t t)
{
  begin = (double) t / CLOCKS_PER_SEC;
  end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "Stopwatch object being destroyed...";
  show();
}

void stopwatch::start()
{
  begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
  end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
  cout << "Elapsed time: " << end - begin;
  cout << "¥n";
}

int main()
{
  stopwatch watch;
  long i;
  watch.start();
  for(i=0; i<320000; i++) ; // forループの時間を測定する
  watch.stop();
  watch.show();

  // 初期値を使用してオブジェクトを作成する
  stopwatch s2(clock());
  for(i=0; i<250000; i++) ; // forループの時間を測定する
  s2.stop();
  s2.show();

  return 0;
}
```

**3.** （省略）

## 5.2：練習問題

**1.** objオブジェクトとtempオブジェクトは通常どおり作成されています．ただし，f()関数によってtempオブジェクトが返されるときに一時オブジェクトが作成され，この一時オブジェクトによってコピーコンストラクタが呼び出されています．

**2.** このプログラムでは，getval()関数にオブジェクトを渡すときに，厳密なコピーが作成されます．getval()関数が終了し，コピーオブジェクトが破棄されると，そのオブジェクトに割り当てられていたメモリ（pが指すメモリ）が解放されます．ただし，getval()関数の呼び出し時に使用したオブジェクトでは，これと同じメモリをまだ必要としています．これを修正したプログラムを次に示します．次のプログラムでは，コピーコンストラクタを使用してこの問題を防いでいます．

```
// 修正したプログラム
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int *p;
public:
  myclass(int i);
  myclass(const myclass &o); // コピーコンストラクタ
  ~myclass() { delete p; }
  friend int getval(myclass o);
};
myclass::myclass(int i)
{
  p = new int;

  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  *p = i;
}

// コピーコンストラクタ
myclass::myclass(const myclass &o)
{
  p = new int; // コピーオブジェクトに専用のメモリを割り当てる

  if(!p) {
    cout << "Allocation error¥n";
```

解答 5

```
    exit(1);
  }
  *p = *o.p;
}

int getval(myclass o)
{
  return *o.p; // 値を取得する
}

int main()
{
  myclass a(1), b(2);

  cout << getval(a) << " " << getval(b);
  cout << "¥n";
  cout << getval(a) << " " << getval(b);

  return 0;
}
```

3. あるオブジェクトを使用して別のオブジェクトを初期化すると，コピーコンストラクタが呼び出されます．通常のコンストラクタは，オブジェクトを作成したときに呼び出されます．

## 5.4：練習問題

1.
```
#include <iostream>
#include <cstdlib>
using namespace std;

long mystrtol(const char *s, char **end, int base =
➡10)
{
  return strtol(s, end, base);
}

int main()
{
  long x;
  char *s1 = "100234";
  char *p;

  x = mystrtol(s1, &p, 16);
  cout << "Base 16: " << x << '¥n';

  x = mystrtol(s1, &p, 10);
  cout << "Base 10: " << x << '¥n';

  x = mystrtol(s1, &p); // デフォルトの基数10を使用する
  cout << "Base 10 by default: " << x << '¥n';

  return 0;
}
```

2. デフォルト引数を受け取るすべての仮引数は，デフォルト引数を受け取らない仮引数よりも右側に書かなければなりません．つまり，デフォルト値を持つ仮引数を書き始めたら，以降の仮引数はすべてデフォルト値を持っていなければなりません．この問題では，qにデフォルト値がありません．

3. カーソル位置制御機能はコンパイラや環境によって異なるので，ここでは解答の一例しか紹介できません．次のプログラムは，コマンドプロンプト環境のBorland C++用に作成したものです．

```
// 注意：このプログラムはBorland C++専用
#include <iostream>
#include <conio.h>
using namespace std;
void myclreol(int len = -1);

int main()
{
  int i;

  gotoxy(1, 1);
  for(i=0; i<24; i++)
    cout <<
"abcdefghijklmnopqrstuvwxyz1234567890¥n";

  gotoxy(1, 2);
  myclreol();
  gotoxy(1, 4);
  myclreol(20);

  return 0;
}
// len仮引数が指定されていない場合は，行末まで消去する
void myclreol(int len)
{
  int x, y;

  x = wherex();  // x位置を取得する
  y = wherey();  // y位置を取得する

  if(len == -1) len = 80-x;

  int i = x;

  for( ; i<=len; i++) cout << ' ';

  gotoxy(x, y); // カーソル位置を再設定する
}
```

4. ほかの仮引数またはローカル変数をデフォルト引数として使うことはできません．

## 5.5：練習問題

1. （省略）

## 5.6：練習問題

1.
```
#include <iostream>
using namespace std;

int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}

int main()
{
  int (*p1)(int, int);
  float (*p2)(float, float);

  p1 = dif; // dif(int, int)のアドレス
  p2 = dif; // dif(float, float);のアドレス

  cout << p1(10, 5) << ' ';
  cout << p2(10.5, 8.9) << '¥n';

  return 0;
}
```

## 第５章：この章の理解度チェック

1.
```
// time_t型の仮引数を受け取るようにdate()関数をオーバーロードする
#include <iostream>
#include <cstdio> // sscanf()関数用にインクルードする
#include <ctime>
using namespace std;

class date {
  int day, month, year;
public:
  date(char *str);
  date(int m, int d, int y) {
    day = d;
    month = m;
    year = y;
  }
  // time_t型の仮引数をオーバーロードする
  date(time_t t);
  void show() {
    cout << month << '/' << day << '/';
    cout << year << '¥n';
  }
};

date::date(char *str)
{
  sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

date::date(time_t t)
{
  struct tm *p;

  p = localtime(&t); // 時刻を各要素に変換する
  day = p->tm_mday;
  month = p->tm_mon;
  year = p->tm_year;
}

int main()
{
```

```
    // 文字列を使用してdateオブジェクトを作成する
    date sdate("12/31/99");

    // 整数を使用してdateオブジェクトを作成する
    date idate(12, 31, 99);

    /* time_t型を使用してdateオブジェクトを作成する
       これによって，システム日付を使用したオブジェクトが作成される */
    date tdate(time(NULL));

    sdate.show();
    idate.show();
    tdate.show();

    return 0;
  }
```

**2.** sampクラスではコンストラクタが1つしか定義されておらず，このコンストラクタでは初期値を必要とします．したがって，初期値を指定せずにsamp方のオブジェクトを宣言するのは正しくありません（つまりsamp xは無効な宣言です）．

**3.** コンストラクタをオーバーロードする理由の1つは，それぞれの状況に合わせて最適なコンストラクタを選択できるように，柔軟性なプログラムを作成することです．また，コンストラクタをオーバーロードすることによって，初期値を持つオブジェクトと初期値を持たないオブジェクトの両方を宣言することもできます．さらに，コンストラクタをオーバーロードして，動的配列を割り当てることもできます．

**4.** コピーコンストラクタの最も広く使われている一般形式は次のとおりです．

```
classname (const classname &obj ) {
    // コンストラクタの本文
}
```

**5.** コピーコンストラクタは，初期化が発生するときに呼び出されます．具体的には，あるオブジェクトを使って明示的にほかのオブジェクトを初期化したとき，関数に仮引数としてオブジェクトを渡したとき，関数からオブジェクトを返し，一時オブジェクトが作成されたときです．

**6.** overloadは古いキーワードです．初期のバージョンのC++では，関数をオーバーロードすることをコンパイラに伝えるために，このキーワードが使われていました．最近のコンパイラでは，このキーワードはサポートされていません．

**7.** デフォルト引数とは，関数を呼び出すときに引数を指定しなかった場合に，対応する関数仮引数として使われる値のことです．

**8.**

```
#include <iostream>
#include <cstring>
using namespace std;

void reverse(char *str, int count = 0);

int main()
{
  char *s1 = "This is a test.";
  char *s2 = "I like C++.";

  reverse(s1);      // 文字列全体を逆順にする
  reverse(s2, 7);   // 最初の7文字を逆順にする

  cout << s1 << '¥n';
  cout << s2 << '¥n';

  return 0;
}

void reverse(char *str, int count)
{
  int i, j;
  char temp;

  if(!count) count = strlen(str)-1;

  for(i=0, j=count; i<j; i++, j--) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
  }
}
```

**9.** デフォルト引数を受け取るすべての仮引数は，デフォルト引数を受け取らない仮引数よりも右側に記述しなければなりません．

**10.** あいまいさは，デフォルトの型変換，参照仮引数，デフォルト引数によって発生することがあります．

**11.** どちらのcompute()関数を呼び出すべきかをコンパイラが判別できないので，このプログラムコードはあいまいです．divisor値としてデフォルト値を使用する最初のcompute()関数を呼び出すべきか，引数を1つだけ受け取る2つ目のcompute()関数を呼び出すべきかを判断できません．

**12.** オーバーロード関数のアドレスを取得する際には，ポインタの型指定によって，使用する関数が決まります．

## 第５章：総合理解度チェック

**1.**

```
#include <iostream>
using namespace std;

void order(int &a, int &b)
{
  int t;

  if(a<b) return;
  else { // aとbを入れ替える
    t = a;
    a = b;
    b = t;
  }
}

int main()
{
  int x=10, y=5;

  cout << "x: " << x << ", y: " << y << "¥n";

  order(x, y);
  cout << "x: " << x << ", y: " << y << "¥n";

  return 0;
}
```

**2.** 参照仮引数を受け取る関数の呼び出し構文は，値仮引数を受け取る関数の構文と同じです．

**3.** デフォルト引数は，関数オーバーロードの省略形として使うことができます．これは結果が同じだからです．
たとえば次の関数があるとします．

```
int f(int a, int b = 0);
```

これは，次の2つのオーバーロード関数を使用するのと機能的に同等です．

```
int f(int a);

int f(int a, int b);
```

**4.**

```
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp() { a = 0; }
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob(88);
  samp obarray[10];
  // ...
}
```

**5.** コピーコンストラクタは，オブジェクトのコピーの作成方法をプログラマが正確に制御したいときに使用します．これは，厳密なコピーを作成すると望ましくない副作用が発生するという場合にのみ重要です．

## 第６章：前章の理解度チェック

**1.**

```
class myclass {
  int x, y;
public:
  myclass(int i, int j) { x=i; y=j; }
  myclass() { x=0; y=0; }
};
```

**2.**

```
class myclass {
  int x, y;
public:
  myclass(int i=0, int j=0) { x=i; y=j; }
};
```

**3.** デフォルト値を持つ引数の後ろに，デフォルト値を持たないパラメータがあってはなりません．

4. 一方が値パラメータを取り，他方が参照パラメータを取るのが唯一の違いであるようなオーバーロードはできません（コンパイラが両者を区別できません）．

5. 頻繁に使われる値が1つ以上あるときは，デフォルト引数を使用するとよいでしょう．使われる可能性が他より大きい値が特にないときは，デフォルト引数を定めてもしかたがありません．

6. いいえ．動的配列を初期化する方法がありません．このクラスにはコンストラクタが1つしかなく，イニシャライザが必要です．

7. コピーコンストラクタは，あるオブジェクトが別のオブジェクトを初期化するときに呼ばれる特殊なコンストラクタです．そのような状況が発生するのは，あるオブジェクトが別のオブジェクトの初期化に明示的に使用されるときか，オブジェクトが関数に引き渡されるときか，関数戻り値として一時オブジェクトが作成されるとき，のいずれかです．

## 6.2 練習問題

1.
```
// *と/をcoordクラスに関してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator*(coord ob2);
  coord operator/(coord ob2);
};

// *をcoordクラスに関してオーバーロードする
coord coord::operator*(coord ob2)
{
  coord temp;

  temp.x = x * ob2.x;
  temp.y = y * ob2.y;

  return temp;
}

// /をcoordクラスに関してオーバーロードする
coord coord::operator/(coord ob2)
{
  coord temp;

  temp.x = x / ob2.x;
  temp.y = y / ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 * o2;
  o3.get_xy(x, y);
  cout << "(o1*o2) X: " << x << ", Y: " << y <<
  ➥"¥n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y <<
  ➥"¥n";

  return 0;
}
```

2. %演算子のオーバーロードが不適切です．その動作は本来の使い方と関連していません．

3. （省略）

## 6.3 練習問題

1.
```
// <と>をcoordクラスに関してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  int operator<(coord ob2);
  int operator>(coord ob2);
};

// <演算子をcoordに関してオーバーロードする
int coord::operator<(coord ob2)
{
  return x<ob2.x && y<ob2.y;
}

// >演算子をcoordに関してオーバーロードする
int coord::operator>(coord ob2)
{
  return x>ob2.x && y>ob2.y;
}

int main()
{
  coord o1(10, 10), o2(5, 3);

  if(o1>o2) cout << "o1 > o2¥n";
  else cout << "o1 <= o2 ¥n";

  if(o1<o2) cout << "o1 < o2¥n";
  else cout << "o1 >= o2¥n";

  return 0;
}
```

## 6.4 練習問題

1.
```
// --をcoordクラスに関してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator--();              // 前置
  coord operator--(int notused); // 後置
};

// 前置--をcoordクラスに関してオーバーロードする
coord coord::operator--()
{
  x--;
  y--;
  return *this;
}

// 後置--をcoordクラスに関してオーバーロードする
coord coord::operator--(int notused)
{
  x--;
  y--;
  return *this;
}

int main()
{
  coord o1(10, 10);
  int x, y;

  o1--; // オブジェクトのデクリメント
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "¥n";

  --o1; // オブジェクトのデクリメント
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "¥n";
  return 0;
}
```

2.
```
// +をcoordクラスに関してオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2); // 2項+
  coord operator+();          // 単項+
};

// +をcoordクラスに関してオーバーロードする
```

```
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// 単項+をcoordクラスに関してオーバーロードする
coord coord::operator+()
{
  if(x<0) x = -x;
  if(y<0) y = -y;
  return *this;
}

int main()
{
  coord o1(10, 10), o2(-2, -2);
  int x, y;
  o1 = o1 + o2; // 加算
  o1.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y <<
  ➡ "¥n";

  o2 = +o2;      // 絶対値
  o2.get_xy(x, y);
  cout << "(+o2) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

## 6.5 練習問題

**1.**

```
/* フレンド関数を使用し,
-と/をcoordクラスに関してオーバーロードする. */
#include <iostream>
using namespace std;

class coord {
  int x, y;      // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator-(coord ob1, coord ob2);
  friend coord operator/(coord ob1, coord ob2);
};

// フレンドを使用し, -をcoordクラスに関してオーバーロードする
coord operator-(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x - ob2.x;
  temp.y = ob1.y - ob2.y;

  return temp;
}

// フレンドを使用し, /をcoordクラスに関してオーバーロードする
coord operator/(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x / ob2.x;
  temp.y = ob1.y / ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 - o2;
  o3.get_xy(x, y);
  cout << "(o1-o2) X: " << x << ", Y: " << y <<
  ➡ "¥n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y <<
  ➡ "¥n";

  return 0;
}
```

**2.**

```
// ob*intでもint*obでも扱えるように*をオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator*(coord ob1, int i);
  friend coord operator*(int i, coord ob2);
};

// *のオーバーロードの1つ目
coord operator*(coord ob1, int i)
{
  coord temp;

  temp.x = ob1.x * i;
  temp.y = ob1.y * i;

  return temp;
}

// *のオーバーロードの2つ目
coord operator*(int i, coord ob2)
{
  coord temp;

  temp.x = ob2.x * i;
  temp.y = ob2.y * i;

  return temp;
}

int main()
{
  coord o1(10, 10), o2;
  int x, y;

  o2 = o1 * 2;  // ob * int
  o2.get_xy(x, y);
  cout << "(o1*2) X: " << x << ", Y: " << y << "¥n";

  o2 = 3 * o1; // int * ob
  o2.get_xy(x, y);
  cout << "(3*o1) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

**3.** フレンド関数を使用すれば，組み込み型を左オペランドにすることができます．メンバ関数を使用するときは，演算子の定義対象になっているクラスのオブジェクトを左オペランドにしなければなりません．

**4.**

```
// フレンドを使用し，coordクラスに関して--をオーバーロードする
#include <iostream>
using namespace std;
class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator--(coord &ob); // 前置
  friend coord operator--(coord &ob, int notused);
                                             // 後置
};

// フレンドを使用し, coordクラスに関して-- (前置) をオーバーロードする
coord operator--(coord &ob)
{
  ob.x--;
  ob.y--;
  return ob;
}

// フレンドを使用し, coordクラスに関して-- (後置) をオーバーロードする
coord operator--(coord &ob, int notused)
{
  ob.x--;
  ob.y--;
  return ob;
}

int main()
{
  coord o1(10, 10);
  int x, y;

  --o1; // オブジェクトのデクリメント
```

解答 6

```
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "¥n";

  o1--; // オブジェクトのデクリメント
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "¥n";

  return 0;
}
```

## 6.6 練習問題

**1.**

```
#include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  int &put(int i);
  int get(int i);
  dynarray &operator=(dynarray &ob);
};

// コンストラクタ
dynarray::dynarray(int s)
{
  p = new int [s];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  size = s;
}

// 要素の格納
int &dynarray::put(int i)
{
  if(i<0 || i>=size) {
    cout << "Bounds error!¥n";
    exit(1);
  }

  return p[i];
}

// 要素の取得
int dynarray::get(int i)
{
  if(i<0 || i>=size) {
    cout << "Bounds error!¥n";
    exit(1);
  }

  return p[i];
}

// =をdynarrayに関してオーバーロードする
dynarray &dynarray::operator=(dynarray &ob)
{
  int i;

  if(size!=ob.size) {
    cout << "Cannot copy arrays of differing
   ➡sizes!¥n";
    exit(1);
  }

  for(i = 0; i<size; i++) p[i] = ob.p[i];
  return *this;
}

int main()
{
  int i;

  dynarray ob1(10), ob2(10), ob3(100);

  ob1.put(3) = 10;
  i = ob1.get(3);
  cout << i << "¥n";

  ob2 = ob1;

  i = ob2.get(3);
  cout << i << "¥n";

  // エラーを生成
  ob1 = ob3; //  !!!
  return 0;
}
```

## 6.7 練習問題

**1.**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *s);
  ~strtype() {
     cout << "Freeing " << (unsigned) p << '¥n';
     delete [] p;
   }
  char *get() { return p; }
  strtype &operator=(strtype &ob);
  char &operator[](int i);
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  len = l;
  strcpy(p, s);
}

// オブジェクトを代入する
strtype &strtype::operator=(strtype &ob)
{
 // さらにメモリが必要かどうかを調べる
  if(len < ob.len) { // さらにメモリを割り当てることが必要
    delete [] p;
    p = new char [ob.len];
    if(!p) {
      cout << "Allocation error¥n";
      exit(1);
    }
  }
  len = ob.len;
  strcpy(p, ob.p);
  return *this;
}

// 文字の添字付け
char &strtype::operator[](int i)
{
  if(i<0 || i>len-1) {
    cout << "¥nIndex value of ";
    cout << i << " is out-of-bounds.¥n";
    exit(1);
  }

  return p[i];
}

int main()
{
  strtype a("Hello"), b("There");

  cout << a.get() << '¥n';
  cout << b.get() << '¥n';

  a = b; // これで，pは上書きされない

  cout << a.get() << '¥n';
  cout << b.get() << '¥n';

  // 配列の添字付けにより，文字にアクセス
  cout << a[0] << a[1] << a[2] << "¥n";

  // 配列の添字付けにより，文字を代入
  a[0] = 'X';
  a[1] = 'Y';
  a[2] = 'Z';

  cout << a.get() << "¥n";
```

```
  return 0;
}
```

2.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  dynarray &operator=(dynarray &ob);
  int &operator[](int i);
};

// コンストラクタ
dynarray::dynarray(int s)
{
  p = new int [s];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }

  size = s;
}

// =をdynarrayに関してオーバーロードする
dynarray &dynarray::operator=(dynarray &ob)
{
  int i;

  if(size!=ob.size) {
    cout << "Cannot copy arrays of differing
  ➡sizes!¥n";
    exit(1);
  }

  for(i = 0; i<size; i++) p[i] = ob.p[i];
  return *this;
}

// []をオーバーロードする
int &dynarray::operator[](int i)
{
  if(i<0 || i>size) {
    cout << "¥nIndex value of ";
    cout << i << " is out-of-bounds.¥n";
    exit(1);
  }
  return p[i];
}

int main()
{
  int i;

  dynarray ob1(10), ob2(10), ob3(100);

  ob1[3] = 10;
  i = ob1[3];
  cout << i << "¥n";

  ob2 = ob1;

  i = ob2[3];
  cout << i << "¥n";

  // エラーを生成
  ob1 = ob3; // 配列のサイズが違う
  return 0;
}
```

## 第6章：この章の理解度チェック

1.
```
// <<と>>をオーバーロードする
#include <iostream>
using namespace std;

class coord {
  int x, y; // 座標値
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator<<(int i);
  coord operator>>(int i);
};

// <<をオーバーロードする
coord coord::operator<<(int i)
{
  coord temp;
  temp.x = x << i;
  temp.y = y << i;
  return temp;
}

// >>をオーバーロードする
coord coord::operator>>(int i)
{
  coord temp;

  temp.x = x >> i;
  temp.y = y >> i;

  return temp;
}

int main()
{
  coord o1(4, 4), o2;
  int x, y;

  o2 = o1 << 2;  // ob << int
  o2.get_xy(x, y);
  cout << "(o1<<2) X: " << x << ", Y: " << y <<
  ➡"¥n";

  o2 = o1 >> 2; // ob >> int
  o2.get_xy(x, y);
  cout << "(o1>>2) X: " << x << ", Y: " << y <<
  ➡"¥n";

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
    i = x; j = y; k = z;
  }
  three_d operator+(three_d ob2);
  three_d operator-(three_d ob2);
  three_d operator++();
  three_d operator--();
};

three_d three_d::operator+(three_d ob2)
{
  three_d temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d three_d::operator++()
{
  x++;
  y++;
  z++;

  return *this;
}

three_d three_d::operator--()
{
  x--;
  y--;
```

```
  z--;

  return *this;
}

int main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3;
  int x, y, z;

  o3 = o1 + o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  o3 = o1 - o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  ++o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  --o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  return 0;
}
```

**3.**

```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  three_d operator+(three_d &ob2);
  three_d operator-(three_d &ob2);
  friend three_d operator++(three_d &ob);
  friend three_d operator--(three_d &ob);
 };

three_d three_d::operator+(three_d &ob2)
{
  three_d temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d &ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d operator++(three_d &ob)
{
  ob.x++;
  ob.y++;
  ob.z++;

  return ob;
}

three_d operator--(three_d &ob)
{
  ob.x--;
  ob.y--;
  ob.z--;

  return ob;
}

int main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3;
  int x, y, z;

  o3 = o1 + o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  o3 = o1 - o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  ++o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  --o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  return 0;
}
```

**4.** 2項メンバ演算子関数には，thisポインタ経由で暗黙的に左オペランドが渡されます．2項フレンド演算子関数には，両方のオペランドが明示的に渡されます．単項メンバ演算子関数には，明示的なパラメータがありません．フレンド単項演算子関数は，パラメータを1つ取ります．

**5.** デフォルトのビット単位コピーでは不十分なときは，=演算子をオーバーロードする必要があります．たとえば，あるオブジェクトのデータの一部だけを別のオブジェクトに代入したいといったことが考えられます．

**6.** いいえ

**7.**

```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  friend three_d operator+(three_d ob, int i);
  friend three_d operator+(int i, three_d ob);
 };

three_d operator+(three_d ob, int i)
{
  three_d temp;
  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

three_d operator+(int i, three_d ob)
{
  three_d temp;

  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

int main()
{
  three_d o1(10, 10, 10);
  int x, y, z;

  o1 = o1 + 10;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "¥n";

  o1 = -20 + o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
```

```
  cout << ", Z: " << z << "¥n";

  return 0;
}
```

**8.**
```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  int operator==(three_d ob2);
  int operator!=(three_d ob2);
  int operator||(three_d ob2);
 };

int three_d::operator==(three_d ob2)
{
  return x==ob2.x && y==ob2.y && z==ob2.z;
}

int three_d::operator!=(three_d ob2)
{
  return x!=ob2.x && y!=ob2.y && z!=ob2.z;
}

int three_d::operator||(three_d ob2)
{
  return x||ob2.x && y||ob2.y && z||ob2.z;
}

int main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3(0, 0, 0);

  if(o1==o1) cout << "o1 == o1¥n";

  if(o1!=o2) cout << "o1 != o2¥n";

  if(o3 || o1) cout << "o1 or o3 is true¥n";

  return 0;
}
```

**9.** []のオーバーロードは通常，クラス内にカプセル化されている配列を，通常の配列添字構文で添字付けできるようにする目的で行われます．

## 第６章：総合理解度チェック

**1.**
```
/*複雑さを避けるため，エラー検査をしていません．
ただし，実際のアプリケーションでこのプログラムコードを使用するときは，
何らかのエラー検査を付け加えなければなりません．*/
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
  char s[80];
public:
  strtype() { *s = '¥0'; }
  strtype(char *p) { strcpy(s, p); }
  char *get() { return s; }
  strtype operator+(strtype s2);
  strtype operator=(strtype s2);
  int operator<(strtype s2);
  int operator>(strtype s2);
  int operator==(strtype s2);
};

strtype strtype::operator+(strtype s2)
{
  strtype temp;

  strcpy(temp.s, s);
  strcat(temp.s, s2.s);

  return temp;
}

strtype strtype::operator=(strtype s2)
{
  strcpy(s, s2.s);

  return *this;
}

int strtype::operator<(strtype s2)
{
  return strcmp(s, s2.s) < 0;
}

int strtype::operator>(strtype s2)
{
  return strcmp(s, s2.s) > 0;
}

int strtype::operator==(strtype s2)
{
  return strcmp(s, s2.s) == 0;
}

int main()
{
  strtype o1("Hello"), o2(" There"), o3;

  o3 = o1 + o2;
  cout << o3.get() << "¥n";

  o3 = o1;
  if(o1==o3) cout << "o1 equals o3¥n";

  if(o1>o2) cout << "o1 > o2¥n";

  if(o1<o2) cout << "o1 < o2¥n";

  return 0;
}
```

## 第７章：前章の理解度チェック

**1.** いいえ．演算子のオーバーロードは，単に演算できるデータ型を拡張するもので，もともとの演算には影響を与えません．

**2.** はい．演算子をC++の組み込み型に関してオーバーロードすることはできません．

**3.** いいえ．優先順位は変更できません．いいえ．オペランドの数は変更できません．

**4.**
```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator+(array ob2);
  array operator-(array ob2);
  int operator==(array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "¥n";
}

array array::operator+(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] + ob2.nums[i];
```

解答
7

```
  return temp;
}

array array::operator-(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] - ob2.nums[i];

  return temp;
}

int array::operator==(array ob2)
{
  int i;

  for(i=0; i<10; i++)
    if(nums[i]!=ob2.nums[i]) return 0;

  return 1;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 equals o2¥n";
  else cout << "o1 does not equal o2¥n";

  if(o1==o3) cout << "o1 equals o3¥n";
  else cout << "o1 does not equal o3¥n";

  return 0;
}
```

**5.**

```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  friend array operator+(array ob1, array ob2);
  friend array operator-(array ob1, array ob2);
  friend int operator==(array ob1, array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "¥n";
}

array operator+(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] + ob2.nums[i];

  return temp;
}

array operator-(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] - ob2.nums[i];

  return temp;
}

int operator==(array ob1, array ob2)
{
  int i;

  for(i=0; i<10; i++)
    if(ob1.nums[i]!=ob2.nums[i]) return 0;

  return 1;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 equals o2¥n";
  else cout << "o1 does not equal o2¥n";

  if(o1==o3) cout << "o1 equals o3¥n";
  else cout << "o1 does not equal o3¥n";

  return 0;
}
```

**6.**

```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator++();
  friend array operator--(array &ob);
};

array::array()
{
  int i;

  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "¥n";
}

// メンバ関数を使って，単項演算子をオーバーロードする
array array::operator++()
{
  int i;

  for(i=0; i<10; i++)
    nums[i]++;
```

```
  return *this;
}

// フレンド関数を使用する
array operator--(array &ob)
{
  int i;

  for(i=0; i<10; i++)
    ob.nums[i]--;

  return ob;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = ++o1;
  o3.show();

  o3 = --o1;
  o3.show();

  return 0;
}
```

**7.** いいえ．代入演算子をオーバーロードするには，メンバ関数を使用しなければなりません．

## 7.1：練習問題

**1.** AとCが正しい文です．

**2.** 基本クラスがpublicとして継承されると，基本クラスの公開メンバは，派生クラスの公開メンバになります．基本クラスの公開メンバがprivateとして継承されると，それは派生クラスの非公開メンバになります．

**3.** （省略）

## 7.2：練習問題

**1.** 基本クラスの被保護メンバがpublicとして継承されると，派生クラスの被保護メンバとなります．privateとして継承されると，派生クラスの非公開メンバになります．protectedとして継承されると，派生クラスの被保護メンバになります．

**2.** 被保護カテゴリが必要とされるのは，基本クラスがあるメンバを非公開にしたまま，派生クラスによるアクセスを許可するためです．

**3.** 変わりません．

## 7.3：練習問題

**1.**

```
#include <iostream>
#include <cstring>
using namespace std;

class mybase {
  char str[80];
public:
  mybase(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

class myderived : public mybase {
  int len;
public:
  myderived(char *s) : mybase(s) {
    len = strlen(s);
  }
  int getlen() { return len; }
  void show() { cout << get() << '¥n'; }
};

int main()
{
  myderived ob("hello");

  ob.show();
  cout << ob.getlen() << '¥n';

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

// 各種車両の基本クラス
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Wheels: " << num_wheels << '¥n';
    cout << "Range: " << range << '¥n';
  }
};

class car : public vehicle {
  int passengers;
public:
  car(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void show()
  {
    showv();
    cout << "Passengers: " << passengers << '¥n';
  }
};

class truck : public vehicle {
  int loadlimit;
public:
  truck(int l, int w, int r) : vehicle(w, r)
  {
    loadlimit = l;
  }
  void show()
  {
    showv();
    cout << "loadlimit " << loadlimit << '¥n';
  }
};

int main()
{
  car c(5, 4, 500);
  truck t(30000, 12, 1200);

  cout << "Car: ¥n";
  c.show();
  cout << "¥nTruck:¥n";
  t.show();

  return 0;
}
```

## 7.4：練習問題

**1.**

```
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```

**2.**

```
#include <iostream>
using namespace std;

class A {
  int i;
public:
  A(int a) { i = a; }
};

class B {
  int j;
public:
  B(int a) { j = a; }
};

class C : public A, public B {
  int k;
public:
  C(int c, int b, int a) : A(a), B(b) {
    k = c;
```

```
    }
  };
```

## 7.5：練習問題

**1.** （省略）

**2.** 派生クラスが2つ以上のクラスを継承し，そのいずれもが同じ基本クラスからの派生クラスであるときは，仮想基本クラスが必要です．仮想基本クラスがないと，同一基本クラスのコピーが2つ以上も最終派生クラスに存在することになりますが，元の基本クラスが仮想基本クラスであれば，最終派生クラスにあるコピーは1つで済みます．

## 第７章：この章の理解度チェック

**1.**
```
#include <iostream>
using namespace std;

class building {
protected:
  int floors;
  int rooms;
  double footage;
};

class house : public building {
  int bedrooms;
  int bathrooms;
public:
  house(int f, int r, double ft, int br, int bth) {
    floors = f;  rooms = r;  footage = ft;
    bedrooms = br; bathrooms = bth;
  }
  void show() {
    cout << "floors: " << floors << '¥n';
    cout << "rooms: " << rooms << '¥n';
    cout << "square footage: " << footage << '¥n';
    cout << "bedrooms: " << bedrooms << '¥n';
    cout << "bathrooms: " << bathrooms << '¥n';
  }
};

class office : public building {
  int phones;
  int extinguishers;
public:
  office(int f, int r, double ft, int p, int ext) {
    floors = f;  rooms = r;  footage = ft;
    phones = p; extinguishers = ext;
  }
  void show() {
    cout << "floors: " << floors << '¥n';
    cout << "rooms: " << rooms << '¥n';
    cout << "square footage: " << footage << '¥n';
    cout << "Telephones: " << phones << '¥n';
    cout << "fire extinguishers: ";
    cout << extinguishers << '¥n';
   }
};

int main()
{
  house h_ob(2, 12, 5000, 6, 4);
  office o_ob(4, 25, 12000, 30, 8);

  cout << "House: ¥n";
  h_ob.show();

  cout << "¥nOffice: ¥n";
  o_ob.show();

  return 0;
}
```

**2.** 基本クラスがpublicとして継承されると，基本クラスの公開メンバは派生クラスの公開メンバになり，基本クラスの非公開メンバは非公開のまま（基本クラス以外には非公開）です．基本クラスがprivateとして継承されると，基本クラスの公開メンバと被保護メンバが派生クラスの非公開メンバになります．

**3.** protectedとして宣言されたメンバは，基本クラス以外には非公開ですが，派生クラスはそれを継承（かつアクセス）できます．継承アクセス指定子としてprotectedを使用すると，基本クラスのすべての公開メンバと被保護メンバが派生クラスの被保護メンバになります．

**4.** コンストラクタは派生順に呼ばれます．デストラクタは，その逆の順序で呼ばれます．

**5.**
```
#include <iostream>
using namespace std;

class planet {
protected:
  double distance;      // 太陽からの距離（マイル数）
  int revolve;          // 日数
public:
  planet(double d, int r) { distance = d; revolve =
r; }
};

class earth : public planet {
  double circumference; // 軌道の円周
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }
  void show() {
    cout << "Distance from sun: " << distance <<
'¥n';
    cout << "Days in orbit: " << revolve << '¥n';
    cout << "Circumference of orbit: ";
    cout << circumference << '¥n';
  }
};

int main()
{
  earth ob(93000000, 365);

  ob.show();

  return 0;
}
```

**6.** このプログラムを修正するには，vehicleを仮想基本クラスとしてmotorizedとroad_useに継承させます．また，この章の「総合理解度チェック」の問1を参照してください．

## 第７章：総合理解度チェック

**1.** コンパイラによっては，インライン関数でswitchを使用できないことがあります．ご使用のコンパイラがそれに該当する場合，関数は自動的に「正規」関数に直されます．

**2.** 継承されない演算子は，唯一，代入演算子です．この理由は簡単です．派生クラスには，基本classにないメンバが含まれますが，基本クラスに関してオーバーロードされた=は，派生クラスで追加されたメンバについては何も知りません．したがって，=はその新しいメンバを正しくコピーできません．

**3.** （省略）

**4.** （省略）

## 第８章：前章の理解度チェック

**1.**
```
#include <iostream>
using namespace std;

class airship {
protected:
  int passengers;
  double cargo;
};

class airplane : public airship {
  char engine;    // プロペラがp，ジェットがj
  double range;
public:
  airplane(int p, double c, char e, double r)
  {
    passengers = p;
    cargo = c;
    engine = e;
    range = r;
  }
  void show();
};

class balloon : public airship {
  char gas;       // 水素がh，ヘリウムがe
  double altitude;
public:
  balloon(int p, double c, char g, double a)
  {
    passengers = p;
    cargo = c;
    gas = g;
```

```
    altitude = a;
  }
  void show();
};

void airplane::show()
{
  cout << "Passengers: " << passengers << '¥n';
  cout << "Cargo capacity: " << cargo << '¥n';
  cout << "Engine: ";
  if(engine=='p') cout << "Propeller¥n";
  else cout << "Jet¥n";
  cout << "Range: " << range << '¥n';
}

void balloon::show()
{
  cout << "Passengers: " << passengers << '¥n';
  cout << "Cargo capacity: " << cargo << '¥n';
  cout << "Gas: ";
  if(gas=='h') cout << "Hydrogen¥n";
  else cout << "Helium¥n";
  cout << "Altitude: " << altitude << '¥n';
}

int main()
{
  balloon b(2, 500.0, 'h', 12000.0);
  airplane b727(100, 40000.0, 'j', 40000.0);

  b.show();
  cout << '¥n';
  b727.show();

  return 0;
}
```

**2.** protectedアクセス指定子は，クラスメンバをそのクラス以外には非公開にします．ただし，派生クラスからはそれにアクセスできます．

**3.** このプログラムは次の出力を表示します．ここから，コンストラクタとデストラクタの呼び出し順序がわかります．

```
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```

**4.** コンストラクタはABCの順序で呼ばれ，デストラクタはCBAの順序で呼ばれます．

**5.**

```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  base(int x, int y) { i = x; j = y; }
  void showij() { cout << i << ' ' << j << '¥n'; }
};

class derived : public base {
  int k;
public:
  derived(int a, int b, int c) : base(b, c) {
    k = a;
  }
  void show() { cout << k << ' '; showij(); }
};

int main()
{
  derived ob(1, 2, 3);

  ob.show();

  return 0;
}
```

**6.** 抜けている単語は，「一般的」と「個別的」です．

## 8.2：練習問題

**1.**

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos);

  cout << -10 << ' ' << 10 << '¥n';

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpoint | ios::uppercase |
            ios::scientific);

  cout << 100.0;

  return 0;
}
```

**3.**

```
#include <iostream>
using namespace std;

int main()
{
  ios::fmtflags f;

  f = cout.flags(); // フラグを格納

  cout.unsetf(ios::dec);
  cout.setf(ios::showbase | ios::hex);
  cout << 100 << '¥n';

  cout.flags(f);    // フラグをリセット

  return 0;
}
```

## 8.3：練習問題

**1.**

```
// 2から100までの自然対数と常用対数の表を作成する
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double x;

  cout.precision(5);
  cout << "          x        log x          ln e¥n¥n";

  for(x = 2.0; x <= 100.0; x++) {
    cout.width(10);
    cout << x << "  ";
    cout.width(10);
    cout << log10(x) << "  ";
    cout.width(10);
    cout << log(x) << '¥n';
  }

  return 0;
}
```

**2.**

```
#include <iostream>
#include <cstring>
using namespace std;

void center(char *s);
int main()
{
  center("Hi there!");
  center("I like C++.");

  return 0;
}

void center(char *s)
{
  int len;

  len = 40+(strlen(s)/2);

  cout.width(len);
  cout << s << '¥n';
}
```

**3.** （省略）

## 8.4：練習問題

**1a.**

```
// 2から100までの自然対数と常用対数の表を作成する
```

解答 8

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x;

  cout << setprecision(5);
  cout << "          x         log x         ln e¥n¥n";

  for(x = 2.0; x <= 100.0; x++) {
    cout << setw(10) << x << "  ";
    cout << setw(10) << log10(x) << "  ";
    cout << setw(10) << log(x) << '¥n';
  }

  return 0;
}
```

**1b.**

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

void center(char *s);

int main()
{
  center("Hi there!");
  center("I like C++.");

  return 0;
}

void center(char *s)
{
  int len;
  len = 40+(strlen(s)/2);

  cout << setw(len) << s << '¥n';
}
```

**2.**

```
cout << setiosflags(ios::showbase | ios::hex) << 100;
```

**3.** 出力ストリームのboolalphaフラグをオンにすると、ブール値がtrueとfalseという単語で表示されます。入力ストリームのboolalphaをオンにすると、trueとfalseという単語でブール値を入力できます。

## 8.5：練習問題

**1.**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype() {delete [] p;}
  friend ostream &operator<<(ostream &stream,
  ➥strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr)+1;
  p = new char [len];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}

ostream &operator<<(ostream &stream, strtype &ob)
{
  stream << ob.p;

  return stream;
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  cout << s1;
  cout << endl << s2 << endl;

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

class planet {
protected:
  double distance;      // 太陽からの距離（マイル数）
  int revolve;          // 日数
public:
  planet(double d, int r) { distance = d; revolve
  ➥= r; }
};

class earth : public planet {
  double circumference; // 軌道の円周
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }

  friend ostream &operator<<(ostream &stream, earth
  ➥ob);
};

ostream &operator<<(ostream &stream, earth ob)
{
  stream << "Distance from sun: " << ob.distance <<
  ➥'¥n';
  stream << "Days in orbit: " << ob.revolve << '¥n';
  stream << "Circumference of orbit: " <<
  ➥ob.circumference;
  stream << '¥n';

  return stream;
}

int main()
{
  earth ob(93000000, 365);

  cout << ob;

  return 0;
}
```

**3.** 挿入子への呼び出しを生成するオブジェクトは、ユーザーが定義したクラスのオブジェクトではありません。そのため、挿入子はメンバ関数になりえません。

## 8.6：練習問題

**1.**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype() {delete [] p;}
  friend ostream &operator<<(ostream &stream,
  ➥strtype &ob);
  friend istream &operator>>(istream &stream,
  ➥strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr)+1;
  p = new char [len];
  if(!p) {
    cout << "Allocation error¥n";
    exit(1);
  }
  strcpy(p, ptr);
}
ostream &operator<<(ostream &stream, strtype &ob)
{
  stream << ob.p;

  return stream;
}

istream &operator>>(istream &stream, strtype &ob)
{
  char temp[255];
```

```
    stream >> temp;

    if(strlen(temp) >= ob.len) {
      delete [] ob.p;
      ob.len = strlen(temp)+1;
      ob.p = new char [ob.len];
      if(!ob.p) {
        cout << "Allocation error¥n";
        exit(1);
      }
    }
    strcpy(ob.p, temp);

    return stream;
  }

  int main()
  {
    strtype s1("This is a test."), s2("I like C++.");

    cout << s1;
    cout << '¥n' << s2;

    cout << "¥nEnter a string: ";
    cin >> s1;
    cout << s1;

    return 0;
  }
```

**2.**

```
#include <iostream>
using namespace std;

class factor {
  int num;   // 数
  int lfact; // 最小因数
public:
  factor(int i);
  friend ostream &operator<<(ostream &stream, factor
➡ob);
  friend istream &operator>>(istream &stream, factor
➡&ob);
};

factor::factor(int i)
{
  int n;

  num = i;

  for(n=2; n < (i/2); n++)
    if(!(i%n)) break;

  if(n<(i/2)) lfact = n;
  else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
  stream >> ob.num;

  int n;

  for(n=2; n < (ob.num/2); n++)
    if(!(ob.num%n)) break;
  if(n<(ob.num/2)) ob.lfact = n;
  else ob.lfact = 1;

  return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
  stream << ob.lfact << " is lowest factor of ";
  stream << ob.num << '¥n';

  return stream;
}

int main()
{
  factor o(32);
  cout << o;

  cin >> o;
  cout << o;

  return 0;
}
```

## 第８章：この章の理解度チェック

**1.**

```
#include <iostream>
using namespace std;

int main()
{
  cout << 100 << ' ';

  cout.unsetf(ios::dec); // decフラグをクリア
  cout.setf(ios::hex);
  cout << 100 << ' ';

  cout.unsetf(ios::hex); // hexフラグをクリア
  cout.setf(ios::oct);
  cout << 100 << '¥n';

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::left);
  cout.precision(2);
  cout.fill('*');
  cout.width(20);

  cout << 1000.5364 << '¥n';

  return 0;
}
```

**3a.**

```
#include <iostream>
using namespace std;

int main()
{
  cout << 100 << ' ';

  cout << hex << 100 << ' ';

  cout << oct << 100 << '¥n';

  return 0;
}
```

**3b.**

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setiosflags(ios::left);
  cout << setprecision(2);
  cout << setfill('*');
  cout << setw(20);

  cout << 1000.5364 << '¥n';

  return 0;
}
```

**4.**

```
ios::fmtflags f;

f = cout.flags();  // 保存

// ...

cout.flags(f);     // 復元
```

**5.**

```
#include <iostream>
using namespace std;

class pwr {
  int base;
  int exponent;
  double result;   // baseのexponent乗
public:
  pwr(int b, int e);
  friend ostream &operator<<(ostream &stream,
➡pwr ob);
  friend istream &operator>>(istream &stream,
➡pwr &ob);
};

pwr::pwr(int b, int e)
{
  base = b;
  exponent = e;
```

解答 8

```
  result = 1;
  for( ; e; e--) result = result * base;
}

ostream &operator<<(ostream &stream, pwr ob)
{
  stream << ob.base << "^" << ob.exponent;
  stream << " is " << ob.result << '¥n';

  return stream;
}

istream &operator>>(istream &stream, pwr &ob)
{
  int b, e;

  cout << "Enter base and exponent: ";
  stream >> b >> e;

  pwr temp(b, e); // 一時オブジェクトの作成

  ob = temp;

  return stream;
}

int main()
{
  pwr ob(10, 2);

  cout << ob;

  cin >> ob;

  cout << ob;

  return 0;
}
```

**6.**

```
// このプログラムは四角形を描く
#include <iostream>
using namespace std;

class box {
  int len;
public:
  box(int l) { len = l; }
  friend ostream &operator<<(ostream &stream,
  ➥box ob);
};

// 四角形を描く
ostream &operator<<(ostream &stream, box ob)
{
  int i, j;

  for(i=0; i<ob.len; i++) stream << '*';
  stream << '¥n';
  for(i=0; i<ob.len-2; i++) {
    stream << '*';
    for(j=0; j<ob.len-2; j++) stream << ' ';
    stream << "*¥n";
  }
  for(i=0; i<ob.len; i++) stream << '*';
  stream << '¥n';

  return stream;
}

int main()
{
  box b1(4), b2(7);

  cout << b1 << endl << b2;

  return 0;
}
```

## 第8章：総合理解度チェック

**1.**

```
#include <iostream>
using namespace std;

#define SIZE 10

// 文字型stackクラスを宣言
class stack {
  char stck[SIZE]; // スタック領域を確保する
  int tos;         // スタック先頭の索引
public:
  stack();
  void push(char ch); // スタックに文字をプッシュする
  char pop();         // スタックから文字をポップする
  friend ostream &operator<<(ostream &stream,
  ➥stack ob);
};

// スタックを初期化する
stack::stack()
{
  tos = 0;
}

// 文字をプッシュする
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full¥n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// 文字をポップする
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty¥n";
    return 0;         // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

ostream &operator<<(ostream &stream, stack ob)
{
  char ch;

  while(ch=ob.pop()) stream << ch;
  stream << endl;

  return stream;
}

int main()
{
  stack s;

  s.push('a');
  s.push('b');
  s.push('c');

  cout << s;
  cout << s;

  return 0;
}
```

**2.**

```
#include <iostream>
#include <ctime>
using namespace std;

class watch {
  time_t t;
public:
  watch() { t = time(NULL); }
  friend ostream &operator<<(ostream &stream,
  ➥watch ob);
};

ostream &operator<<(ostream &stream, watch ob)
{
  struct tm *localt;

  localt = localtime(&ob.t);
  stream << asctime(localt) << endl;
  return stream;
}

int main()
{
  watch w;

  cout << w;

  return 0;
}
```

**3.**

```
#include <iostream>
using namespace std;

class ft_to_inches {
```

```
  double feet;
  double inches;
public:
  void set(double f) {
    feet = f;
    inches = f * 12;
  }
  friend istream &operator>>(istream &stream,
                             ft_to_inches &ob);
  friend ostream &operator<<(ostream &stream,
                             ft_to_inches ob);
};

istream &operator>>(istream &stream, ft_to_inches
➡&ob)
{
  double f;

  cout << "Enter feet: ";
  stream >> f;
  ob.set(f);

  return stream;
}

ostream &operator<<(ostream &stream, ft_to_inches
➡ob)
{
  stream << ob.feet << " feet is " << ob.inches;
  stream << " inches¥n";
  return stream;
}

int main()
{
  ft_to_inches x;

  cin >> x;
  cout << x;

  return 0;
}
```

## 第９章：前章の理解度チェック

**1.**
```
#include <iostream>
using namespace std;

int main()
{
  cout.width(40);
  cout.fill(':');

  cout << "C++ is fun" << '¥n';

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

int main()
{
  cout.precision(4);
  cout << 10.0/3.0 << '¥n';

  return 0;
}
```

**3.**
```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setprecision(4) << 10.0/3.0 << '¥n';

  return 0;
}
```

**4.** 挿入子とは，オーバーロードされたoperator<<()関数です．クラスのデータを出力ストリームに出力します．抽出子とは，オーバーロードされたoperator>>()関数です．クラスのデータを入力ストリームから入力します．

**5.**
```
#include <iostream>
using namespace std;

class date {
  char d[9]; // 日付をmm/dd/yyという文字列として格納する
public:
  friend ostream &operator<<(ostream &stream,
  ➡date ob);
  friend istream &operator>>(istream &stream,
  ➡date &ob);
};

ostream &operator<<(ostream &stream, date ob)
{
  stream << ob.d << '¥n';

  return stream;
}

istream &operator>>(istream &stream, date &ob)
{
  cout << "Enter date (mm/dd/yy): ";
  stream >> ob.d;

  return stream;
}

int main()
{
  date ob;

  cin >> ob;
  cout << ob;

  return 0;
}
```

**6.** パラメータをとるマニピュレータを使用するには，プログラムに<iomanip>をインクルードしなければなりません．

**7.** cin，cout，cerr，clogの各定義済みストリームです．

## 9.1：練習問題

**1.**
```
// 時刻と日付を示す
#include <iostream>
#include <ctime>
using namespace std;

// 時刻と日付の出力マニピュレータ
ostream &td(ostream &stream)
{
  struct tm *localt;
  time_t t;

  t = time(NULL);
  localt = localtime(&t);
  stream << asctime(localt) << endl;

  return stream;
}

int main()
{
  cout << td << '¥n';

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

// 16進出力をオンにし，Xを大文字にする
ostream &sethex(ostream &stream)
{
  stream.unsetf(ios::dec | ios::oct);
  stream.setf(ios::hex | ios::uppercase |
              ios::showbase);
  return stream;
}

// フラグのリセット
ostream &reset(ostream &stream)
{
  stream.unsetf(ios::hex | ios::uppercase |
                ios::showbase);
  stream.setf(ios::dec);
  return stream;
}

int main()
{
  cout << sethex << 100 << '¥n';
  cout << reset << 100 << '¥n';

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

// 10文字をスキップする
istream &skipchar(istream &stream)
{
  int i;
  char c;

  for(i=0; i<10; i++) stream >> c;

  return stream;
}

int main()
{
  char str[80];

  cout << "Enter some characters: ";
  cin >> skipchar >> str;

  cout << str << '¥n';

  return 0;
}
```

## 9.2：練習問題

1.
```
// テキストファイルをコピーし，コピーした文字の数を表示する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <input> <output>¥n";
    return 1;
  }

  ifstream fin(argv[1]);   // 入力ファイルのオープン
  ofstream fout(argv[2]);  // 出力ファイルの作成

  if(!fin) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  if(!fout) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  char ch;
  unsigned count=0;

  fin.unsetf(ios::skipws); // スペースをスキップしない
  while(!fin.eof()) {
    fin >> ch;
    if(!fin.eof()) {
      fout << ch;
      count++;
    }
  }

  cout << "Number of bytes copied: " << count <<
  ➥ '¥n';

  fin.close();
  fout.close();

  return 0;
}
```

このプログラムが表示する結果が，ディレクトリをとったときに示される値と異なるときは，文字変換が起こっていると考えられます．特に，キャリッジリターン/ラインフィード（CR/LF）文字列は，読み取られると改行に変換されます．出力時には改行は1文字として数えられるものの，再びCR/LF文字列に変換し直されます．

2.
```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream pout("phone");

  if(!pout) {
    cout << "Cannot open PHONE file.¥n";
    return 1;
  }
  pout << "Isaac Newton  415 555-3423¥n";
  pout << "Robert Goddard 213 555-2312¥n";
  pout << "Enrico Fermi  202 555-1111¥n";

  pout.close();

  return 0;
}
```

3.
```
// 語数
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: COUNT <input>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  int count=0;
  char ch;

  in >> ch;              // 最初の非スペース文字を見つける

  // 最初の非スペース文字を見つけてからは，スペースをスキップしない
  in.unsetf(ios::skipws); // スペースをスキップしない

  while(!in.eof()) {
    in >> ch;
    if(isspace(ch)) {
      count++;
      while(isspace(ch) && !in.eof()) in >> ch;
    }
  }

  cout << "Word count: " << count << '¥n';

  in.close();

  return 0;
}
```

4. is_open()関数は，呼び出しを行ったストリームがオープンファイルにリンクされているとき，真を返します．

## 9.3：練習問題

1a.
```
// ファイルをコピーし，コピーした文字の数を表示する
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <input> <output>¥n";
    return 1;
  }

  ifstream fin(argv[1], ios::in | ios::binary);
                              // 入力ファイルをオープン
  ofstream fout(argv[2], ios::out | ios::binary);
                              // 出力ファイルを作成

  if(!fin) {
    cout << "Cannot open input file¥n";
    return 1;
  }

  if(!fout) {
    cout << "Cannot open output file¥n";
    return 1;
  }

  char ch;
  unsigned count=0;

  while(!fin.eof()) {
    fin.get(ch);
    if(!fin.eof()) {
      fout.put(ch);
      count++;
```

```
    }
  }

  cout << "Number of bytes copied: " << count <<
➥'¥n';

  fin.close();
  fout.close();

  return 0;
}
```

**1b.**
```
// 語数
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: COUNT <input>¥n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  int count=0;
  char ch;

  // 最初の非スペース文字を見つける
  do {
    in.get(ch);
  } while(isspace(ch));

  while(!in.eof()) {
    in.get(ch);
    if(isspace(ch)) {
      count++;
      while(isspace(ch) && !in.eof()) in.get(ch);
                                          // 次の語を見つける
    }
  }

  cout << "Word count: " << count << '¥n';

  in.close();

  return 0;
}
```

**2.**
```
// 挿入子を使って，account情報をファイルに出力する
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class account {
  int custnum;
  char name[80];
  double balance;
public:
  account(int c, char *n, double b)
  {
    custnum = c;
    strcpy(name, n);
    balance = b;
  }
  friend ostream &operator<<(ostream &stream,
  ➥account ob);
};

ostream &operator<<(ostream &stream, account ob)
{
  stream << ob.custnum << ' ';
  stream << ob.name << ' ' << ob.balance;
  stream << '¥n';

  return stream;
}

int main()
{
  account  Rex(1011, "Ralph Rex", 12323.34);
  ofstream out("accounts", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  out << Rex;

  out.close();

  return 0;
}
```

## 9.4：練習問題

**1.**
```
// get()を使用して，スペースを含む文字列を読み取る

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char str[80];

  cout << "Enter your name: ";
  cin.get(str, 79);

  cout << str << '¥n';

  return 0;
}
```

このプログラムは，get()とgetline()のどちらを使用してもはたらきは変わりません．

**2.**
```
// getline()を使用して，テキストファイルを表示する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: PR <filename>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char str[255];

  while(!in.eof()) {
    in.getline(str, 254);
    cout << str << '¥n';
  }

  in.close();

  return 0;
}
```

**3.** 本文の解説を参照．

## 9.5：練習問題

**1.**
```
// ファイルを逆方向に画面に表示する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: REVERSE <filename>¥n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char ch;
  long i;
```

```
  // ファイルの終わり（eof文字の手前）に行く
  in.seekg(0, ios::end);
  i = (long) in.tellg(); // ファイルに何バイトあるか調べる
  i -= 2;                // eofの前まで後退

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

**2.**
```
// ファイル中の文字の入れ替え
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: SWAP <filename>¥n";
    return 1;
  }

  // ファイルを入出力用にオープン
  fstream io(argv[1], ios::in | ios::out |
➥ios::binary);
  if(!io) {
    cout << "Cannot open file.¥n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ; !io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    io.put(ch2);
    io.put(ch1);
  }

  io.close();

  return 0;
}
```

## 9.6：練習問題

**1a.**
```
/*ファイルを逆方向に画面に表示し,
エラー検査を追加する*/
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: REVERSE <filename>¥n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file¥n";
    return 1;
  }

  char ch;
  long i;

  // ファイルの終わり（eof文字の手前）に行く
  in.seekg(0, ios::end);
  if(!in.good()) return 1;
  i = (long) in.tellg(); // ファイルに何バイトあるか調べる
  if(!in.good()) return 1;
  i -= 2;                // eofの前まで後退

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    if(!in.good()) return 1;
    in.get(ch);
    if(!in.good()) return 1;
    cout << ch;
  }

  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

**1b.**
```
// ファイル中の文字を入れ替え, エラー検査を追加する
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: SWAP <filename>¥n";
    return 1;
  }

  // ファイルを入出力用にオープン
  fstream io(argv[1], ios::in | ios::out |
➥ios::binary);

  if(!io) {
    cout << "Cannot open file.¥n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ;!io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(!io.good()) return 1;
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.put(ch2);
    if(!io.good()) return 1;
    io.put(ch1);
    if(!io.good()) return 1;
  }

  io.close();
  if(!io.good()) return 1;

  return 0;
}
```

## 9.7：練習問題

**1.** （省略）

## 第９章：この章の理解度チェック

**1.**
```
#include <iostream>
using namespace std;

ostream &tabs(ostream &stream)
{
  stream << '¥t' << '¥t' << '¥t' ;
  stream.width(20);

  return stream;
}

int main()
{
  cout << tabs << "Testing¥n";

  return 0;
}
```

**2.**
```
#include <iostream>
#include <cctype>
using namespace std;

istream &findalpha(istream &stream)
{
  char ch;

  do {
    stream.get(ch);
```

```
  } while(!isalpha(ch));
  return stream;
}

int main()
{
  char str[80];

  cin >> findalpha >> str;
  cout << str << '¥n';

  return 0;
}
```

**3.**
```
// ファイルをコピーし，大文字小文字を反対にする
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: COPYREV <source> <target>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
    }
  };

  in.close();
  out.close();

  return 0;
}
```

**4.**
```
// 文字数をかぞえる
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: COUNT <source>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  // alpha[]の初期化
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(isalpha(ch)) {    // 文字が見つかったら，数に入れる
      ch = toupper(ch); // 正規化
      alpha[ch-'A']++;  // 'A'-'A' == 0, 'B'-'A'
                        // == 1, など
    }
  };

  // 発生回数の表示
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] <<
    ➡ '¥n';
  }

  in.close();

  return 0;
}
```

**5a.**
```
/*ファイルをコピーして，大文字小文字を反対にする.
エラー検査あり*/
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: COPYREV <source> <target>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Cannot open output file";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.good() && !in.eof()) return 1;
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
      if(!out.good()) return 1;
    }
  };

  in.close();
  out.close();
  if(!in.good() && !out.good()) return 1;
  return 0;
}
```

**5b.**
```
// 文字を数える．エラー検査を行う．
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: COUNT <source>¥n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  // alpha[]の初期化
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(!in.good() && !in.eof()) return 1;
    if(isalpha(ch)) {    // 文字が見つかったら，数に入れる
```

解答 9

```
      ch = toupper(ch); // 正規化
      alpha[ch-'A']++;  // 'A'-'A' == 0, 'B'-'A' ==
      ➥1, など
    }
  };

  // 発生回数の表示
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] <<
    ➥'¥n';
  }

  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

**6.** getポインタの位置を指定するにはseekg()，putポインタの位置を指定するにはseekp()を使います．

## 第9章：総合理解度チェック

**1.**

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

#define SIZE 40

class inventory {
  char item[SIZE]; // 品名
  int onhand;      // 在庫数量
  double cost;     // 原価
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  void store(fstream &stream);
  void retrieve(fstream &stream);
  friend ostream &operator<<(ostream &stream,
  ➥inventory ob);
  friend istream &operator>>(istream &stream,
  ➥inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " on hand at $" << ob.cost << '¥n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
  cout << "Enter item name: ";
  stream >> ob.item;
  cout << "Enter number on hand: ";
  stream >> ob.onhand;
  cout << "Enter cost: ";
  stream >> ob.cost;

  return stream;
}

void inventory::store(fstream &stream)
{
  stream.write(item, SIZE);
  stream.write((char *) &onhand, sizeof(int));
  stream.write((char *) &cost, sizeof(double));
}

void inventory::retrieve(fstream &stream)
{
  stream.read(item, SIZE);
  stream.read((char *) &onhand, sizeof(int));
  stream.read((char *) &cost, sizeof(double));
}

int main()
{

  fstream inv("inv", ios::out | ios::binary);
  int i;

  inventory pliers("pliers", 12, 4.95);
  inventory hammers("hammers", 5, 9.45);
  inventory wrenches("wrenches", 22, 13.90);
  inventory temp("", 0, 0.0);

  if(!inv) {
    cout << "Cannot open file for output.¥n";
    return 1;
  }
  // ファイルへ書き出す
  pliers.store(inv);
  hammers.store(inv);
  wrenches.store(inv);
  inv.close();

  // 入力用にオープン
  inv.open("inv", ios::in | ios::binary);

  if(!inv) {
    cout << "Cannot open file for input.¥n";
    return 1;
  }

  do {
    cout << "Record # (-1 to quit): ";
    cin >> i;
    if(i == -1) break;
    inv.seekg(i*(SIZE+sizeof(int)+sizeof(double)),
    ➥ios::beg);
    temp.retrieve(inv);
    cout << temp;
  } while(inv.good());

  inv.close();

  return 0;
}
```

**2.** （省略）

## 第10章：前章の理解度チェック

**1.**

```
#include <iostream>
using namespace std;

ostream &setsci(ostream &stream)
{
  stream.setf(ios::scientific | ios::uppercase);

  return stream;
}

int main()
{
  double f = 123.23;

  cout << setsci << f;
  cout << '¥n';

  return 0;
}
```

**2.**

```
// コピーし，タブをスペースに置き換える
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <in> <out>¥n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  ofstream out(argv[2]);
  if(!out) {
    cout << "Cannot open output file.¥n";
    return 1;
  }

  char ch;
  int i = 8;

  while(!in.eof()) {
    in.get(ch);
    if(ch=='¥t') for( ; i>0; i--) out.put(' ');
    else out.put(ch);
```

```
    if(i == -1 || ch=='¥n') i = 8;
    i--;
  }

  in.close();
  out.close();

  return 0;
}
```

3.
```
// ファイル内検索
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: SEARCH <file> <word>¥n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "Cannot open input file.¥n";
    return 1;
  }

  char str[255];
  int count=0;

  while(!in.eof()) {
    in >> str;
    if(!strcmp(str, argv[2])) count++;
  }

  cout << argv[2] << " found " << count;
  cout << " number of times.¥n";

  in.close();

  return 0;
}
```

4. 次の文でできます.
```
out.seekp(233, ios::beg);
```

5. rdstate()、good()、eof()、fail()、bad()です.

6. C++入出力システムは、プログラマが独自に作成したクラスを扱えるようカスタマイズできます.

## 10.1：練習問題

1. （省略）

## 10.2：練習問題

1.
```
#include <iostream>
using namespace std;

class num {
public:
  int i;
  num(int x) { i = x; }
  virtual void shownum() { cout << i << '¥n'; }
};

class outhex : public num {
public:
  outhex(int n) : num(n) {}
  void shownum() { cout << hex << i << '¥n'; }
};

class outoct : public num {
public:
  outoct(int n) : num(n) {}
  void shownum() { cout << oct << i << '¥n'; }
};

int main()
{
  outoct o(10);
  outhex h(20);

  o.shownum();
  h.shownum();

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class dist {
public:
  double d;
  dist(double f) { d = f; }
  virtual void trav_time()
  {
    cout << "Travel time at 60 mph: ";
    cout << d / 60 << '¥n';
  }
};

class metric : public dist {
public:
  metric(double f) : dist(f) {}
  void trav_time()
  {
    cout << "Travel time at 100 kph: ";
    cout << d / 100 << '¥n';
  }
};

int main()
{
  dist *p, mph(88.0);
  metric kph(88);

  p = &mph;
  p->trav_time();

  p = &kph;
  p->trav_time();

  return 0;
}
```

## 10.3：練習問題

1. （省略）

2. 抽象クラスは、定義上、少なくとも1つの純粋仮想関数を含んでいるクラスです. つまり、その関数には、このクラスに関する本体が存在しません. クラス定義が不完全であるため、オブジェクトを作成しようにも作成できません.

3. derived1に関してfunc()を呼んだとき、使用されるのはbase内のfunc()です. このしくみが機能するのは、仮想関数が階層的であるためです.

## 10.4：練習問題

1.
```
// 仮想関数の実例
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
  list *head;  // リスト先頭へのポインタ
  list *tail;  // リスト末尾へのポインタ
  list *next;  // 次項目へのポインタ
  int num;     // 格納される値
  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// キュー型リストの作成
class queue : public list {
public:
  void store(int i);
  int retrieve();
};

void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // リスト末尾に置く
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
```

解答 10

```
  if(!head) head = tail;
}
int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

// スタック型リストの作成
class stack : public list {
public:
  void store(int i);
  int retrieve();
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // スタック様の操作になるよう，リスト最前部に置く
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リストの先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

// ソート済みリストの作成
class sorted : public list {
public:
  void store(int i);
  int retrieve();
};

void sorted::store(int i)
{
  list *item;
  list *p, *p2;

  item = new sorted;
  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);

  }
  item->num = i;

  // 次項目の置き場所を見つける
  p = head;
  p2 = NULL;
  while(p) { // 中へ
    if(p->num > i) {
      item->next = p;
      if(p2) p2->next = item;  // 先頭要素でない
      if(p==head) head = item; // 新しい先頭要素
      break;
    }
    p2 = p;
    p = p->next;
  }
  if(!p) {                       // 終わりへ
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
  }
  if(!head) // 先頭要素
    head = item;
}

int sorted::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

int main()
{
  list *p;

  // キューの実演
  queue q_ob;
  p = &q_ob; // キューを指し示す

  p->store(1);
  p->store(2);
  p->store(3);
  cout << "Queue: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '¥n';

  // スタックの実現
  stack s_ob;
  p = &s_ob; // スタックを指し示す

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Stack: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '¥n';

  // ソート済みリストの実演
  sorted sorted_ob;
  p = &sorted_ob;

  p->store(4);
  p->store(1);
  p->store(3);
  p->store(9);
  p->store(5);

  cout << "Sorted: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '¥n';


  return 0;
}
```

**2.** （省略）

## 第10章：この章の理解度チェック

1. 仮想関数とは本質的にはプレースホルダ関数であり，基本クラスで宣言され，その基本クラスから派生したクラスで再定義されます．この再定義をオーバーライドと言います．

2. 非メンバ関数とコンストラクタ関数は，仮想関数になれません．

3. 仮想関数は，基本クラスポインタを使って実行時ポリモーフィズムをサポートします．基本クラスポインタが，仮想関数を含む派生クラスのオブジェクトを指し示していると，指し示されているオブジェクトの型によって，呼ばれる関数が決まります．

4. 純粋仮想関数とは，基本クラスに関する定義をまったく含まない関数です．

5. 抽象クラスとは，少なくとも1つの純粋仮想関数を含んでいる基本クラスです．ポリモーフィッククラスとは，少なくとも1つの仮想関数を含んでいるクラスです．

6. このコードは正しくありません．仮想関数の再定義では，戻り型，およびパラメータの型と数が，もとの関数と同じでなければなりません．この場合，f()の再定義で，パラメータの数が違っています．

7. はい．

8. （省略）

## 第10章：総合理解度チェック

1.
```
// 仮想関数の実例
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
  list *head;  // リスト先頭へのポインタ
  list *tail;  // リスト末尾へのポインタ
  list *next;  // 次項目へのポインタ
  int num;     // 格納される値

  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// キュー型リストの作成
class queue : public list {
public:
  void store(int i);
  int retrieve();
  queue operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // リスト末尾に置く
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;
  return i;
}

// スタック型リストの作成
class stack : public list {
public:
  void store(int i);
  int retrieve();
  stack operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Allocation error.¥n";
    exit(1);
  }
  item->num = i;

  // スタックのような操作になるよう，リストの最前部に置く
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.¥n";
    return 0;
  }

  // リスト先頭から取り除く
  i = head->num;
  p = head;
  head = head->next;
  delete p;
  return i;
}

int main()
{
  // キューの実演
  queue q_ob;

  q_ob + 1;
  q_ob + 2;
  q_ob + 3;

  cout << "Queue: ";
  cout << q_ob--;
  cout << q_ob--;
  cout << q_ob--;

  cout << '¥n';

  // スタックの実演
  stack s_ob;

  s_ob + 1;
  s_ob + 2;
  s_ob + 3;

  cout << "Stack: ";
  cout << s_ob--;
  cout << s_ob--;
  cout << s_ob--;

  cout << '¥n';

  return 0;
}
```

2. オーバーロードされた関数どうしの間では，パラメータの数または型が必ず違っていなければなりません．オーバーライドされた仮想関数は，元の関数とプロトタイプが同一（つまり，戻り型が同じで，パラメータの型と数も同じ）でなければなりません．

3. （省略）

## 第11章：前章の理解度チェック

1. 仮想関数とは，基本クラスでvirtualキーワードを使って宣言され，派生クラスでオーバーライドされる関数のことです．

2. 純粋仮想関数とは，基本クラスで本体が定義されていない関数のことです．つまり，派生クラスで関数を上書きしなければなりません．少なく

とも1つの純粋仮想関数を含む基本クラスのことを抽象クラスと呼びます．

**3.** 空欄に当てはまる語は「仮想」と「基本」です．

**4.** 非純粋仮想関数を派生クラスで上書きしない場合，派生クラスは基本クラスの仮想関数を使用します．

**5.** 実行時ポリモーフィズムを行う主な利点は，柔軟性が得られることです．主な欠点は実行速度が低下することです．

## 11.1：練習問題

**1.** （省略）

**2.**

```
#include <iostream>
using namespace std;

template <class X> X min(X a, X b)
{
  if(a<=b) return a;
  else return b;
}

int main()
{
  cout << min(12.2, 2.0);
  cout << endl;
  cout << min(3, 4);
  cout << endl;
  cout << min('c', 'a');
  return 0;
}
```

**3.**

```
#include <iostream>
#include <cstring>
using namespace std;

template <class X> int find(X object, X *list,
➥int size)
{
  int i;

  for(i=0; i<size; i++)
    if(object == list[i]) return i;
  return -1;
}

int main()
{
  int a[] = {1, 2, 3, 4};
  char *c = "this is a test";
  double d[] = {1.1, 2.2, 3.3};

  cout << find(3, a, 4);
  cout << endl;
  cout << find('a', c, (int) strlen(c));
  cout << endl;
  cout << find(0.0, d, 3);

  return 0;
}
```

**4.** 汎用関数を使うと，さまざまなデータ型に適用できる汎用のアルゴリズムを定義できます（つまり，データ型ごと特定のアルゴリズムを手作業で明示的に作成する必要がありません）．さらに，汎用関数はC++プログラミングの共通テーマである「1つのインターフェイスで複数のメソッド」という概念を実現するのにも役立ちます．

## 11.2：練習問題

**1.** （省略）

**2.**

```
// 汎用キューを作成する
#include <iostream>
using namespace std;

#define SIZE 100

template <class Qtype> class q_type {
  Qtype queue[SIZE]; // キューを保存する
  int head, tail;    // 先頭と末尾の索引
public:
  q_type() { head = tail = 0; }
  void q(Qtype num); // 保存する
  Qtype deq();       // 取得する
};

// キューに値をプットする
template <class Qtype> void q_type<Qtype>::q(Qtype
➥num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full.¥n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // 循環する
  queue[tail] = num;
}

// キューから値を削除する
template <class Qtype> Qtype q_type<Qtype>::deq()
{
  if(head == tail) {
    cout << "Queue is empty.¥n";
    return 0;              // またはその他のエラーインジケータ
  }
  head++;
  if(head==SIZE) head = 0; // 循環する
  return queue[head];
}

int main()
{
  q_type<int> q1;
  q_type<char> q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i-1+'A');
  }

  for(i=1; i<=10; i++) {
    cout << "Dequeue 1: " << q1.deq() << "¥n";
    cout << "Dequeue 2: " << q2.deq() << "¥n";
  }
  return 0;
}
```

**3.**

```
#include <iostream>
using namespace std;

template <class X> class input {
  X data;
public:
  input(char *s, X min, X max);
  // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
{
  do {
    cout << s << ": ";
    cin >> data;
  } while( data < min || data > max);
}

int main()
{
  input<int> i("enter int", 0, 10);
  input<char> c("enter char", 'A', 'Z');

  return 0;
}
```

## 11.3：練習問題

**1.** （省略）

**2.** tryブロックを通る前にthrow文が呼び出されています．

**3.** 文字例外が投げられますが，catch文では文字ポインタしか処理しません（つまり，文字例外を処理するための対応するcatch文がありません）．

**4.** 対応するcatch文のない例外が投げられると，terminate()関数が呼び出されてプログラムが異常終了します．

## 11.4：練習問題

**1.** （省略）

**2.** throwに対応するcatch文がありません．

**3.** 1つの解決法としては，catch(int)ハンドラを作成する方法があります．もう1つの解決法としては，catch(...)ハンドラを作成してすべて例外を捕獲する方法があります．

**4.** すべての例外を捕獲する文はcatch(...)です．

**5.**
```
#include <iostream>
#include <cstdlib>
using namespace std;

double divide(double a, double b)
{
  try {
    if(!b) throw(b);
  }
  catch(double) {
    cout << "Cannot divide by zero.¥n";
    exit(1);
  }
  return a/b;
}

int main()
{
  cout << divide(10.0, 2.5) << endl;
  cout << divide(10.0, 0.0);

  return 0;
}
```

## 11.5：練習問題

**1.** デフォルトでは，new演算子は割り当てエラーが発生したときに例外を投げます．これに対して，new(nothrow)は，メモリ割り当てエラーが発生したときにヌルポインタを返します．

**2.**
```
p = new(nothrow) int;

if(!p) {
  cout << "Allocation error.¥n";
  // ...
}

try {
  p = new int;
} catch(bad_alloc ba) {
  cout << "Allocation error.¥n";
  // ...
}
```

## 第11章：この章の理解度チェック

**1.**
```
#include <iostream>
#include <cstring>
using namespace std;

// 汎用のモード検索関数
template <class X> X mode(X *data, int size)
{
  register int t, w;
  X md, oldmd;
  int count, oldcount;

  oldmd = 0;
  oldcount = 0;
  for(t=0; t<size; t++) {
    md = data[t];
    count = 1;
    for(w = t+1; w < size; w++)
      if(md==data[w]) count++;
    if(count > oldcount) {
      oldmd = md;
      oldcount = count;
    }
  }
  return oldmd;
}

int main()
{
  int i[] = { 1, 2, 3, 4, 2, 3, 2, 2, 1, 5};
  char *p = "this is a test";

  cout << "mode of i: " << mode(i, 10) << endl;
  cout << "mode of p: " << mode(p, (int) strlen(p));

  return 0;
}
```

**2.**
```
#include <iostream>
using namespace std;

template <class X> X sum(X *data, int size)
{
  int i;
  X result = 0;

  for(i=0; i<size; i++) result += data[i];

  return result;
}

int main()
{
  int i[] = {1, 2, 3, 4};
  double d[] = {1.1, 2.2, 3.3, 4.4};

  cout << sum(i, 4) << endl;
  cout << sum(d, 4) << endl;

  return 0;
}
```

**3.**
```
#include <iostream>
using namespace std;

// 汎用のバブルソート
template <class X> void bubble(X *data, int size)
{
  register int a, b;
  X t;

  for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--)
      if(data[b-1] > data[b]) {
        t = data[b-1];
        data[b-1] = data[b];
        data[b] = t;
      }
}

int main()
{
  int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
  double d[] = {1.2, 5.5, 2.2, 3.3};
  int j;

  bubble(i, 10); // intをソートする
  bubble(d, 4);  // doubleをソートする

  for(j=0; j<10; j++) cout << i[j] << ' ';
  cout << endl;

  for(j=0; j<4; j++) cout << d[j] << ' ';
  cout << endl;

  return 0;
}
```

**4.**
```
/* この関数では，2つの値を保持する汎用の
   スタックを作成する */
#include <iostream>
using namespace std;

#define SIZE 10

// 汎用のスタッククラスを作成する
template <class StackType> class stack {
  StackType stck[SIZE][2]; // スタック領域を確保する
  int tos;                 // スタック先頭の索引

public:
  void init() { tos = 0; }
  void push(StackType ob, StackType ob2);
  StackType pop(StackType &ob2);
};

// オブジェクトをプッシュする
template <class StackType>
void stack<StackType>::push(StackType ob, StackType
ob2)
{
  if(tos==SIZE) {
    cout << "Stack is full.¥n";
    return;
  }
  stck[tos][0] = ob;
  stck[tos][1] = ob2;
  tos++;
}

// オブジェクトをポップする
template <class StackType>
StackType stack<StackType>::pop(StackType &ob2)
{
  if(tos==0) {
```

解答 11

```
    cout << "Stack is empty.¥n";
    return 0;              // スタックが空の場合はヌルを返す
  }
  tos--;
  ob2 = stck[tos][1];
  return stck[tos][0];
}

int main()
{
  // 文字スタックの例
  stack<char> s1, s2;      // 2つのスタックを作成する
  int i;
  char ch;

  // スタックを初期化する
  s1.init();
  s2.init();

  s1.push('a', 'b');
  s2.push('x', 'z');
  s1.push('b', 'd');
  s2.push('y', 'e');
  s1.push('c', 'a');
  s2.push('z', 'x');

  for(i=0; i<3; i++) {
    cout << "Pop s1: " << s1.pop(ch);
    cout << ' ' << ch << "¥n";
  }
  for(i=0; i<3; i++) {
    cout << "Pop s2: " << s2.pop(ch);
    cout << ' ' << ch << "¥n";
  }

  // doubleスタックの例
  stack<double> ds1, ds2;  // 2つのスタックを作成する
  double d;

  // スタックを初期化する
  ds1.init();
  ds2.init();

  ds1.push(1.1, 2.0);
  ds2.push(2.2, 3.0);
  ds1.push(3.3, 4.0);
  ds2.push(4.4, 5.0);
  ds1.push(5.5, 6.0);
  ds2.push(6.6, 7.0);

  for(i=0; i<3; i++) {
    cout << "Pop ds1: " << ds1.pop(d);
    cout << ' '<< d << "¥n";
  }

  for(i=0; i<3; i++) {
    cout << "Pop ds2: " << ds2.pop(d);
    cout << ' '<< d << "¥n";
  }

  return 0;
}
```

**5.** try，catch，throwの一般形式は次のとおりです．

```
try {
  // tryブロック
  throw exp;
}
catch(type arg) {
  // ...
}
```

**6.**

```
/* この関数では，例外処理を含む
   汎用スタックを作成する */
#include <iostream>
using namespace std;

#define SIZE 10

// 汎用スタッククラスを作成する
template <class StackType> class stack {
  StackType stck[SIZE];     // スタック領域を確保する
  int tos;                  // スタック先頭の索引

public:
  void init() { tos = 0; } // スタックを初期化する
  void push(StackType ch); // スタックにオブジェクトを
                           // プッシュする
  StackType pop();       // スタックからオブジェクトをポップする
};

// オブジェクトをプッシュする
template <class StackType>
void stack<StackType>::push(StackType ob)
{
  try {
    if(tos==SIZE) throw SIZE;
  }
  catch(int) {
    cout << "Stack is full.¥n";
    return;
  }
  stck[tos] = ob;
  tos++;
}

// オブジェクトをポップする
template <class StackType>
StackType stack<StackType>::pop()
{
  try {
    if(tos==0) throw 0;
  }
  catch(int) {
    cout << "Stack is empty.¥n";
    return 0;              // スタックが空の場合はヌルを返す
  }
  tos--;
  return stck[tos];
}

int main()
{
  // 文字スタックの例
  stack<char> s1, s2;      // 2つのスタックを作成する
  int i;
  // スタックを初期化する
  s1.init();
  s2.init();

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop()
➡<< "¥n";
  for(i=0; i<4; i++) cout << "Pop s2: " << s2.pop()
➡<< "¥n";

  // doubleスタックの例
  stack<double> ds1, ds2;  // 2つのスタックを作成する

  // スタックを初期化する
  ds1.init();
  ds2.init();

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " <<
➡ds1.pop() << "¥n";
  for(i=0; i<4; i++) cout << "Pop ds2: " <<
➡ds2.pop() << "¥n";

  return 0;
}
```

**7.** （省略）

**8.** 割り当てエラーが発生してnew演算子が例外を投げた場合は，何らかの方法で必ずエラーを処理することができます（単にプログラムを異常終了することもあります）．これに対して，ヌルポインタを返すことによって報告される割り当てエラーは，この点をプログラムコードでチェックしないと見過ごされてしまう可能性があります．問題となるのはプログラムでヌルポインタを使おうとしたときで，プログラムはしばらくは正常に動作するかもしれませんが，やがて誤動作を始め，最終的には予測不可能な（繰り返しが不可能な）方法でクラッシュします．これは診断の困難な種類のバグです．

## 第11章：総合理解度チェック

**1.** （省略）

**2.** （省略）

## 第12章：前章の理解度チェック

**1.** C++の汎用関数とは，さまざまなデータ型に適用できる汎用的な一連の動作のことです．汎用関数を作成するにはtemplateキーワードを使用します．汎用関数の一般形式は次のとおりです．

```
template <class Ttype> ret-type func-name(param-list)
{
    // ...
}
```

**2.** C++の汎用クラスとは，そのクラスに関するすべての動作を定義するものです．ただし，実際のオブジェクトはそのクラスのオブジェクトを作成する際に，仮引数として指定します．汎用クラスの一般形式は次のとおりです．

```
template <class Ttype> class class-name
{
    // ...
}
```

**3.**
```
#include <iostream>
using namespace std;

// aをbに返す
template <class X> X gexp(X a, X b)
{
   X i, result=1;

  for(i=0; i<b; i++) result *= a;

  return result;
}

int main()
{
  cout << gexp(2, 3) << endl;
  cout << gexp(10.0, 2.0);

  return 0;
}
```

**4.**
```
#include <iostream>
#include <fstream>
using namespace std;

template <class CoordType> class coord {
  CoordType x, y;
public:
  coord(CoordType i, CoordType j) { x = i; y = j; }
  void show() { cout << x << ", " << y << endl; }
};

int main()
{
  coord<int> o1(1, 2), o2(3, 4);

  o1.show();
  o2.show();

  coord<double> o3(0.0, 0.23), o4(10.19, 3.098);

  o3.show();
  o4.show();

  return 0;
}
```

**5.** 例外を監視したいすべての文をtryブロックに含めます．例外が発生したら，throw文を使用してその例外を投げ，対応するcatch文でそれを処理します．

**6.** できません．

**7.** terminate()関数は，対応するcatch文のない例外が投げられたときに呼び出されます．unexpected()関数は，関数のthrow句内で指定されていない例外を投げようとしたときに投げられます．

**8.** すべての型の例外を処理できるcatch文の構文は，catch(...)です．

## 12.1：練習問題

**1.** C++では，基本クラスポインタが指すオブジェクトの型，または基本クラス参照が参照するオブジェクトの型がコンパイル時にはわからないことがあるので，RTTIが必要となります．

**2.** BaseClassクラスをポリモーフィッククラスではないようにした場合，出力は次のようになります．

```
Typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass
```

**3.** 正しい．

**4.** `if(typeid(*p) == typeid(D2)) ...`

**5.** 偽です．同じテンプレートクラスを使っていますが，それぞれが使用するデータの型が異なります．

**6.** （省略）

## 12.2：練習問題

**1.** dynamic_cast演算子は，ポリモーフィック型変換の有効性を判別します．

**2.**
```
#include <iostream>
#include <typeinfo>
using namespace std;

class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
};

class D2: public B {
  void f() {}
};

int main()
{
  B *p;
  D2 ob;

  p = dynamic_cast<D2 *> (&ob);

  if(p) cout << "Cast OK";
  else cout << "Cast Fails";

  return 0;
}
```

**3.**
```
int main()
{
  int i;
  Shape *p;

  for(i=0; i<10; i++) {
    p = generator(); // 次のオブジェクトを取得する

    cout << typeid(*p).name() << endl;

    // NullShapeではない場合にだけオブジェクトを描画する
    if(!dynamic_cast<NullShape *> (p))
      p->example();
  }

  return 0;
}
```

**4.** 有効ではありません．BpとDpは，それぞれ基本的に異なるオブジェクトを指すポインタです．

## 12.3：練習問題

**1.** 新しいキャスト演算子を使うと，より安全で明示的な方法で型変換を行うことができます．

**2.**
```
#include <iostream>
using namespace std;

void f(const double &i)
{
  double &v = const_cast<double &> (i);

  v = 100.0;
}

int main()
{
  double x = 98.6;

  cout << x << endl;
  f(x);
  cout << x << endl;
```

```
  return 0;
}
```

3. const_castはconst指定子を上書きするので，オブジェクトに対して予想外の修正を加える可能性が生じます．

## 第12章：この章の理解度チェック

1. typeid演算子は，type_info型クラスのオブジェクトへの参照を返します．このオブジェクトには型情報が含まれています．

2. typeidを使うには，<typeinfo>をインクルードする必要があります．

3. 新しい型変換演算子を次に示します．

| 演算子 | 目的 |
|---|---|
| dynamic_cast | ポリモーフィック型変換を行う |
| reinterpret_cast | ある型のポインタをほかの型に変換する |
| static_cast | 「通常の」型変換を行う |
| const_cast | const指定を取り除く |

4.
```
#include <iostream>
#include <typeinfo>
using namespace std;

class A {
  virtual void f() {}
};

class B : public A {
};

class C: public B {
};

int main()
{
  A *p, a_ob;
  B b_ob;
  C c_ob;
  int i;

  cout << "Enter 0 for A objects, ";
  cout << "1 for B objects or ";
  cout << "2 for C objects.¥n";

  cin >> i;

  if(i==1) p = &b_ob;
  else if(i==2) p = &c_ob;
  else p = &a_ob;

  if(typeid(*p) == typeid(A))
    cout << "A object";
  if(typeid(*p) == typeid(B))
    cout << "B object";
  if(typeid(*p) == typeid(C))
    cout << "C object";

  return 0;
}
```

5. typeidを使ってポリモーフィック型変換の有効性を判別する際には，dynamic_cast演算子をtypeidの代わりとして使うことができます．

6. typeid演算子は，type_info型オブジェクトへの参照を返します．

## 第12章：総合理解度チェック

1. 次に，例外処理を使用して，割り当てエラーを監視するgenerator()関数を示します．

```
// 例外処理を使用して割り当てエラーをチェックする
Shape *generator()
{
  try {
    switch(rand() % 4) {
      case 0:
        return new Line;
      case 1:
        return new Rectangle;
      case 2:
        return new Triangle;
      case 3:
        return new NullShape;
    }
  } catch (bad_alloc ba) {
    return NULL;
  }
  return NULL;
}
```

2. 次に，new(nothrow)を使うように修正したgenerator()関数を示します．

```
// new(nothrow)を使用する
Shape *generator()
{
  Shape *temp;

  switch(rand() % 4) {
    case 0:
      temp = new(nothrow) Line;
      break;
    case 1:
      temp = new(nothrow) Rectangle;
      break;
    case 2:
      temp = new(nothrow) Triangle;
      break;
    case 3:
      temp = new(nothrow) NullShape;
  }

  if(temp) return temp;
  else return NULL;
}
```

3. （省略）

## 第13章：前章の理解度チェック

1. C++形式の型変換に加えて，次のキャスト演算子を使用できます．

| 演算子 | 目的 |
|---|---|
| dynamic_cast | ポリモーフィック型変換を行う |
| reinterpret_cast | ある型のポインタをほかの型に変換する |
| static_cast | 「通常の」型変換を行う |
| const_cast | const指定を取り除く |

2. type_infoはデータ型に関する情報をカプセル化したクラスです．typeid演算子からはtype_infoオブジェクトへの参照が返されます．

3. オブジェクトの型を判別する演算子はtypeidです．

4.
```
if(typeid(Derived) == typeid(*p))
  cout << "p points to derived object¥n";
else
  cout << "p points to a Base object¥n";
```

5. 下線部に当てはまる語は「派生」です．

6. できません．

### 13.1：練習問題

1.
```
/* 「using namespace std」文を使わずに
     空白を|に変換する */
#include <iostream>
#include <fstream>

int main(int argc, char *argv[])
{
  if(argc!=3) {
    std::cout << "Usage: CONVERT <input>
    ➥<output>¥n";
    return 1;
  }

  std::ifstream fin(argv[1]);     // 入力ファイルを開く
  std::ofstream fout(argv[2]);    // 出力ファイルを開く

  if(!fout) {
    std::cout << "Cannot open output file.¥n";
    return 1;
  }
  if(!fin) {
    std::cout << "Cannot open input file.¥n";
    return 1;
  }

  char ch;

  fin.unsetf(std::ios::skipws);  // 空白を飛ばさない
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '|';
    if(!fin.eof()) fout << ch;
```

```
  }

  fin.close();
  fout.close();

  return 0;
}
```

**2.** 無名名前空間を使うと，宣言された識別子のスコープをそのファイル内に限定することができます．

**3.** 次の形式のusingでは，指定したメンバだけを現在の名前空間に取り込みます．

```
using name::member ;
```

これに対し，次の形式のusingでは，名前空間全体を取り込みます．

```
using namespace name;
```

**4.** ストリームcinやcoutを含むC++標準ライブラリは，std名前空間内で定義されているので，ほとんどのプログラムでは便宜上std名前空間を取り込みます．これによって，std::という修飾子を付けることなく，標準ライブラリ名を直接使用できるようになります．ほとんどのプログラムでは標準ライブラリへのすべての参照にstd::を付けるという代替法を利用できます．また，std::cinまたはstd::coutだけのusing文を作成するという代替法もあります．

**5.** ライブラリコードを独自の名前空間に含めることにより，名前が競合する可能性を減らすことができます．

## 13.2：練習問題

**1.**

```
// 文字列型を整数型に変換する
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len =
  ➡strlen(s); }
  operator char *() { return str; }
  operator int() { return len; }
};

int main()
{
  strtype s("Conversion functions are convenient.");
  char *p;
  int l;

  l = s; // sを整数（文字列の長さ）に変換する
  p = s; // sをchar *（文字列を指すポインタ）に変換する

  cout << "The string:¥n";
  cout << p << "¥nis " << l << " chars long.¥n";

  return 0;
}
```

**2.**

```
#include <iostream>
using namespace std;

int p(int base, int exp);

class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
  operator int() { return p(base, exp); }
};

// 基数のexp乗を返す
Return base to the exp power.
int p(int base, int exp)
{
  int temp;

  for(temp=1; exp; exp--) temp = temp * base;

  return temp;
}

 int main()
{
  pwr o1(2, 3), o2(3, 3);
  int result;

  result = o1;
  cout << result << '¥n';

  result = o2;
  cout << result << '¥n';

  // 次のようにして，cout文内で直接使用できる
  cout << o1+100 << '¥n';

  return 0;
}
```

## 13.3：練習問題

**1.**

```
// 出力を追跡するリソース共有の例
#include <iostream>
#include <cstring>
using namespace std;

class output {
  static char outbuf[255]; // これは共有リソース
  static int inuse;         // 0の場合はバッファを利用できる．
                            // それ以外の場合は使用中
  static int oindex;        // outbufの索引
  char str[80];
  int i;     // str内の次のcharの索引
  int who;   // オブジェクトを識別する．0より大きくなければならない
public:
  output(int w, char *s) { strcpy(str, s); i = 0;
  ➡who = w; }

  /* この関数は，バッファを待機している場合は-1を返す
     出力を完了した場合は0，
     バッファを使用中の場合はwhoを返す
  */
  int putbuf()
  {
    if(!str[i]) {          // 出力完了
      inuse = 0;           // バッファを解放する
      return 0;            // 終了を通知する
    }
    if(!inuse) inuse = who; // バッファを取得する
    if(inuse != who) {
      cout << "Process " << who << " Currently
      ➡blocked¥n";
      return -1;           // ほかのオブジェクトが使用中
    }
    if(str[i]) { // まだ出力する文字がある
      outbuf[oindex] = str[i];
      cout << "Process " << who << " sending
      ➡char¥n";
      i++; oindex++;
      outbuf[oindex] = '¥0'; // 常にヌルで終了する
      return 1;
    }
    return 0;
  }
  void show() { cout << outbuf << '¥n'; }
};

char output::outbuf[255];  // これは共有リソース
int output::inuse = 0;     // 0の場合はバッファを利用できる．
それ以外の場合は使用中
int output::oindex = 0;    // outbufの索引

int main()
{
  output o1(1, "This is a test"), o2(2, " of
statics");

  while(o1.putbuf() | o2.putbuf()) ; // 文字を出力する

  o1.show();

  return 0;
}
```

**2.**

```
#include <iostream>
#include <new>
using namespace std;

class test {
  static int count;
public:
  test() { count++; }
  ~test() { count--; }
  int getcount() { return count; }
};

int test::count = 0;
```

解答 13

```
int main()
{
  test o1, o2, o3;

  cout << o1.getcount() << " objects in existence\n";

  test *p;

  /* 古い形式と新しい形式の両方のエラー処理を使用して,
     割り当てエラーを監視する */
  try {
    p = new test;          // オブジェクトを割り当てる
    if(!p) {               // 古い形式
      cout << "Allocation error\n";
      return 1;
    }
  } catch(bad_alloc ba) { // 新しい形式
      cout << "Allocation error\n";
      return 1;
  }

  cout << o1.getcount();
  cout << " objects in existence after allocation\n";

  // オブジェクトを削除する
  delete p;

  cout << o1.getcount();
  cout << " objects in existence after deletion\n";

  return 0;
}
```

## 13.4：練習問題

**1.** このプログラムを修正するには，currentをmutableにして，constメンバ関数であるcounting()関数内からcurrentを修正できるようにするだけです．修正済みのプログラムを次に示します．

```
// 修正後のプログラム
#include <iostream>
using namespace std;

class CountDown {
  int incr;
  int target;
  mutable int current;     // currentをmutableにする
public:
  CountDown(int delay, int i=1) {
    target = delay;
    incr = i;
    current = 0;
  }

  bool counting() const {
    current += incr;
    if(current >= target) {
      cout << "\a";
      return false;
    }
    cout << current << " ";
    return true;
  }
};

int main()
{
  CountDown ob(100, 2);

  while(ob.counting()) ;

  return 0;
}
```

**2.** できません．constメンバから非constメンバ関数を呼び出すことができるとしたら，非const関数を使って呼び出し元オブジェクトを修正できてしまいます．

## 13.5：練習問題

**1.** 暗黙的な変換が行われます．

**2.** このプログラムコードは有効です．これは，C++によってintからdoubleへの自動変換が定義されるからです．

**3.** 暗黙的なコンストラクタ変換に伴う1つの問題として，そのような変換が発生していることを忘れてしまう可能性があります．たとえば，代入文によって発生する暗黙的な変換は，外見上はオーバーロード代入演算子とよく似ています．ただし，その動作は同じであるとは限りません．他人が使用するクラスを作成する場合には，そのクラスを使う側の御用を防ぐために，暗黙的なコンストラクタ変換を禁止した方がよいでしょう．

## 13.6：練習問題

**1.** （省略）

## 13.7：練習問題

**1.**
```
/* bufに書き込まれた文字数
   を表示する
*/
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[255];

  ostrstream ostr(buf, sizeof buf);

  ostr << "Array-based I/O uses streams just like ";
  ostr << "'normal' I/O\n" << 100;
  ostr << ' ' << 123.23 << '\n';

  // マニピュレータを使うこともできる
  ostr << hex << 100 << ' ';
  // 書式フラグも使用できる
  ostr.setf(ios::scientific);

  ostr << dec << 123.23;
  ostr << endl << ends;   // バッファを確実に
                          // ヌル終了にする

  // 結果の文字列を表示する
  cout << buf;

  cout << ostr.pcount();

  return 0;
}
```

**2.**
```
/* 配列ベースの入出力を使用して,
   1つの配列の内容をもう1つの配列にコピーする
*/
#include <iostream>
#include <strstream>
using namespace std;

char inbuf[] = "This is a test of C++ array-based
➥I/O";
char outbuf[255];

int main()
{
  istrstream istr(inbuf);
  ostrstream ostr(outbuf, sizeof outbuf);

  char ch;

  while(!istr.eof()) {
    istr.get(ch);
    if(!istr.eof()) ostr.put(ch);
  }
  ostr.put('\0');         // ヌルで終了する
  cout << "Input: " << inbuf << '\n';
  cout << "Output: " << outbuf << '\n';

  return 0;
}
```

**3.**
```
// 文字列を浮動少数点数に変換する
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  float f;
  char s[] = "1234.564"; // 文字列として表現された浮動少数点数

  istrstream istr(s);

  // 内部表現に変換する単純な方法
  istr >> f;

  cout << "Converted form: " << f << '\n';

  return 0;
}
```

## 第13章：この章の理解度チェック

1. 通常のメンバ変数の場合は各オブジェクトがそれぞれ独自のコピーを持ちますが，staticメンバ変数のコピーは1つしか存在しません．このコピーはそのクラスのすべてのオブジェクトによって共有されます．

2. 配列ベースの入出力を使用するには，<strstream>ヘッダをインクルードします．

3. ありません．

4. `extern "C" int counter();`

5. 変換関数は，オブジェクトを単にほかの型と互換性のある値に変換します．変換関数は，オブジェクトを組み込みデータ型と互換性のある値に変換する場合に使用するのが一般的です．

6. explicitキーワードはコンストラクタにしか使用できません．explicitキーワードを使うと，暗黙的なコンストラクタ変換を防ぐことができます．

7. constメンバ関数からは，呼び出し元のオブジェクトを修正することができません．

8. 名前空間はnamespaceキーワードを使って宣言し，宣言領域を定義します．

9. mutableキーワードを使うと，constメンバ関数からそのクラスのデータメンバを修正できるようになります．

## 第13章：総合理解度チェック

1. あります．暗黙的な変換によって実行する操作が，コンストラクタの仮引数の型に対するオーバーロード代入演算子によって実行される操作と同じである場合は，代入演算子をオーバーロードする必要はありません．

2. できます．

3. 新しいライブラリを独自の名前空間に含め，ほかのプログラムコードとの名前の競合を防ぐことができます．この利点は，新しいライブラリを使用して古いプログラムコードを更新した場合にも当てはまります．

4. （省略）

## 第14章：前章の理解度チェック

1. 名前空間は，識別子名を局所化し，名前の競合を避けるためにC++に追加されました．サードパーティ製クラスライブラリが増加したため，名前の競合の問題が深刻化しました．

2. メンバ関数をconstとして指定するには，関数の仮引数リストの後にconstキーワードを続けます．次に例を示します．

```
int f(int a) const;
```

3. 正しくない．mutableを使うと，constメンバ関数からメンバ変数を変更できるようになります．

4.
```
class X {
  int a, b;
public:
  X(int i, int j) { a = i, b = j; }
  operator int() { return a+b; }
};
```

5. 正しい．

6. 有効ではありません．明示的な指定子を使用すると，int型からDemo型への自動変換が行われません．

### 14.1：練習問題

1. コンテナとは，ほかのオブジェクトを含むオブジェクトのことです．アルゴリズムとは，コンテナの内容を操作するルーチンのことです．反復子はポインタと似ています．

2. 2項と単項．

3. 5種類の反復子とは，ランダムアクセス，双方向，前方，入力，出力です．

### 14.3：練習問題

1. （省略）

2. ベクトルに含まれているオブジェクトにはすべてデフォルトコンストラクタを持っていなければなりません．

3.
```
// Coordオブジェクトをベクトルに保存する
#include <iostream>
#include <vector>
using namespace std;

class Coord {
public:
  int x, y;
  Coord() { x = y = 0; }
  Coord(int a, int b) { x = a; y = b; }
};

bool operator<(Coord a, Coord b)
{
  return (a.x+a.y) < (b.x+b.y);
}

bool operator==(Coord a, Coord b)
{
  return (a.x+a.y) == (b.x+b.y);
}

int main()
{
  vector<Coord> v;
  int i;

  for(i=0; i<10; i++)
    v.push_back(Coord(i, i));

  for(i=0; i<v.size(); i++)
    cout << v[i].x << "," << v[i].y << " ";

  cout << endl;

  for(i=0; i<v.size(); i++)
    v[i].x = v[i].x * 2;

  for(i=0; i<v.size(); i++)
    cout << v[i].x << "," << v[i].y << " ";

  return 0;
}
```

### 14.4：練習問題

1. （省略）

2.
```
// リストの基本操作
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst;      // 空のリストを作成する
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size = " << lst.size() << endl;
  list<char>::iterator p;
  cout << "Contents: ";
  for(i=0; i<lst.size(); i++) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
    lst.push_back(*p); // リストの最後に要素を追加する
  }

  cout << endl;

  if(!lst.empty())
    cout << "List is not empty.¥n";

  return 0;
}
```

このプログラムからの出力を次に示します．

```
Size = 10
Contents: ABCDEFGHIJ
List is not empty.
```

このプログラムでは，要素を前方から削除し，後方に追加しています．したがって，リストは空になりません．リストを表示するループは，size()を使用してリストのサイズを取得することによって制御しています．

3.
```
// 2つのプロジェクトリストを結合する
#include <iostream>
```

```
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
  char name[40];
  int days_to_completion;
  Project() {
    strcpy(name, "");
    days_to_completion = 0;
  }
  Project(char *n, int d) {
    strcpy(name, n);
    days_to_completion = d;
  }

  void add_days(int i) {
    days_to_completion += i;
  }

  void sub_days(int i) {
    days_to_completion -= i;
  }

  bool completed() { return !days_to_completion; }

  void report() {
    cout << name << ": ";
    cout << days_to_completion;
    cout << " days left.¥n";
  }
};

bool operator<(const Project &a, const Project &b)
{
  return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
  return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
  return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
  return a.days_to_completion != b.days_to_completion;
}

int main()
{
  list<Project> proj;
  list<Project> proj2;

  proj.push_back(Project("Compiler", 35));
  proj.push_back(Project("Spreadsheet", 190));
  proj.push_back(Project("STL Implementation",
  ➥1000));

  proj2.push_back(Project("Database", 780));
  proj2.push_back(Project("Mail Merge", 50));
  proj2.push_back(Project("COM Objects", 300));

  proj.sort();
  proj2.sort();

  proj.merge(proj2); // プロジェクトを結合する

  list<Project>::iterator p = proj.begin();
  /* プロジェクトを表示する */
  while(p != proj.end()) {
    p->report();
    p++;
  }

  return 0;
}
```

## 14.5：練習問題

**1.** （省略）

**2.**

```
// 名前と電話番号のマップ
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
  char str[20];
public:
  name() { strcpy(str, ""); }
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

// nameオブジェクトに対する「より小」を定義しなければならない
bool operator<(name a, name b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class phonenum {
  char str[20];
public:
  phonenum() { strcmp(str, ""); }
  phonenum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<name, phonenum> m;
  // 名前と電話番号をマップに格納する
  m.insert(pair<name,
    phonenum>(name("Joe"), phonenum("555-4323")));
  m.insert(pair<name,
    phonenum>(name("Tom"), phonenum("555-9784")));
  m.insert(pair<name,
    phonenum>(name("Larry"), phonenum("212
    ➥555-9372")));
  m.insert(pair<name,
    phonenum>(name("Tod"), phonenum("01 11
    ➥232-4232")));

  // 名前をもとに電話番号を検索する
  char str[80];
  cout << "Enter name: ";
  cin >> str;

  map<name, phonenum>::iterator p;

  p = m.find(name(str));
  if(p != m.end())
    cout << "Phone Number: " <<  p->second.get();
  else
    cout << "Name not in map.¥n";

  return 0;
}
```

**3.** <演算子を定義する必要があります.

## 14.6：練習問題

**1.**

```
// sort()アルゴリズムを使用してベクトルをソートする
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
  vector<char> v;
  int i;

  // 無作為の文字のベクトルを作成する
  for(i=0; i<10; i++)
    v.push_back('A'+ (rand()%26));

  cout << "Original contents: ";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";

  cout << endl << endl;

  // ベクトルをソートする
  sort(v.begin(), v.end());

  cout << "Sorted contents: ";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";

  return 0;
}
```

2.
```
// merge()アルゴリズムを使用して，2つのリストを結合する
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
  list<char> lst1, lst2, lst3(20);
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Contents of lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // 2つのリストを結合する
  merge(lst1.begin(), lst1.end(),
        lst2.begin(), lst2.end(),
        lst3.begin());

  cout << "Merged list:¥n";
  p = lst3.begin();
  while(p != lst3.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

## 14.7：練習問題

1.
```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
  list<string> str;

  str.push_back(string("one"));
  str.push_back(string("two"));
  str.push_back(string("three"));
  str.push_back(string("four"));
  str.push_back(string("five"));
  str.push_back(string("six"));
  str.push_back(string("seven"));
  str.push_back(string("eight"));
  str.push_back(string("nine"));
  str.push_back(string("ten"));

  str.sort(); // ベクトルをソートする

  list<string>::iterator p = str.begin();
  while(p != str.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

2.
```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str;

  cout << "Enter a string: ";
  cin >> str;

  int i = count(str.begin(), str.end(), 'e');

  cout << i << " of them are e.¥n";

  return 0;
}
```

3.
```
#include <iostream>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

int main()
{
  string str;

  cout << "Enter a string: ";
  cin >> str;

  int i = count_if(str.begin(), str.end(), islower);

  cout << i << " of them are lowercase.¥n";

  return 0;
}
```

4. stringはbasic_stringテンプレートクラスを特化したオブジェクトです．

## 第14章：この章の理解度チェック

1. STLを使うと，デバッグ済みの数多くの共通データ構造体とアルゴリズムを手軽に利用することができます．STLクラスはテンプレート化されているので，どのデータ型にも使用できます．

2. コンテナとはほかのオブジェクトを格納するオブジェクトのことです．反復子はポインタのようなものです．アルゴリズムはコンテナの内容を操作するものです．

3.
```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{
  vector <int> v(10);
  list<int> lst;
  int i;

  for(i=0; i<10; i++) v[i] = i;

  for(i=0; i<10; i++)
    if(!(v[i]%2)) lst.push_back(v[i]);

  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

4. string型を使うと，演算子を使って文字列を操作することができます．ただし，stringオブジェクトの操作は，ヌル終端文字配列の操作と比べて効率がよくありません．

5. 条件式とは，真または偽を返す関数のことです．

6. （省略）

7. （省略）

8. （省略）

## 第14章：総合理解度チェック

1. （省略）

2. （省略）

3. （省略）

4. （省略）

# 索　引

## 記号

## オ



## カ



## キ



## ク



## ケ



## コ

## サ



## シ



## ス



## セ



## ソ



## タ



## チ



## テ



## ト



## ナ



## ニ

## リ



## レ



## ロ

## ● 著者紹介

**Herbert Schildt（ハーバート・シルト）**

C/C++ 言語の世界的な屈指の権威であり，Windows プログラミングの大家でもある．C/C++ 言語に関する優れたプログラミング書の執筆者として知られ，『独習 C』，『C/C++ プログラマのための Windows95 プログラミング』，『C プログラマのための C++ エッセンシャル』，『Windows 98 プログラミング』，『標準講座 C++』（いずれも小社刊）ほか多数の著書は，さまざまな外国語に翻訳され，全世界で200万部以上が読まれている．ANSI C と C++ の標準化委員会のメンバー．Universal Computing Laboratories,Inc. 社長．イリノイ大学で修士号を取得．

**独習 C++（シープラスプラス） 改訂版**

CD-ROM　1 枚付き

1999年 4 月30日　初版第 1 刷発行
2001年10月 5 日　初版第 9 刷発行

| | |
|---|---|
| 著　　者 | Herbert Schildt（ハーバート・シルト） |
| 翻　　訳 | 有限会社 トップスタジオ |
| 監　　修 | 神林 靖（かんばやし・やすし） |
| 発 行 人 | 速水 浩二 |
| 発 行 所 | 株式会社 翔泳社 |
| 装　　丁 | 岩波 路恵 |
| 編集・組版 | 有限会社 トップスタジオ |
| 印刷・製本 | 株式会社 廣済堂 |



ISBN4-88135-701-8　　Printed in Japan

# 独習シリーズ

## 独習C 改訂版 CD-ROM 1枚付き

ハーバート・シルト著　ANSI標準準拠
SE編集部 訳
柏原 正三 監修
定価:本体3,200円+税
ISBN4-88135-700-X

定評あるC言語自習書.ANSI C標準に準拠した最も汎用性の高いコーディングスタイルを採用しています.徹底した演習を行う学習メソッドにより,C言語の基礎を効果的かつ確実に身につけることができます.まったく曖昧さが残らない完全な独習のできる教材です.

## 独習Java CD-ROM 1枚付き

ジョゼフ・オニール著
トップスタジオ 訳
武藤 健志 監修
定価:本体3,600円+税
ISBN4-88135-748-4

Javaの教科書として好評発売中.最新バージョンのJava 2に対応しています.プログラミング初心者をはじめ,他の言語の経験をお持ちの方にもお薦めします.オブジェクト指向言語の一つの実装系であるJavaの基本がよくわかり,厳選された例題と徹底した演習により応用力を身に付けることができます.付属のCD-ROMには,Java 2(JDK 1.2)を収録.この1冊があれば,すぐにJavaの学習を始めることができます.

## 独習Visual Basic 6.0
CD-ROM 2枚付き

ジョン・ソチャ,ダン・ラウメル,デブラ・ホール著
トップスタジオ 訳/編
佐藤立子,竹内里佳 監修
定価:本体3,200円+税
ISBN4-88135-733-6
特別付録●Visual Basic 6.0日本語評価版

VB解説書の定番として実績ある「独習Visual Basic」の三訂版.最新バージョン6.0に完全対応しました.「実践→発見→わかる」という三段階のアプローチで,無理なくWindowsプログラミングの基本を学べます.CD-ROMには学習に完全な機能を備えたVisual Basic 6.0日本語評価版*が収録されていますから,購入したその日から最新の環境でプログラミングをスタートできます.(*実行ファイルの作成とオンラインヘルプを除くすべての機能が無期限で使えます)

**独習C++**
**改訂版**

■利用分類
プログラミング
■製品分類
C++
■レベル
初級　中級　上級

ISBN4-88135-701-8
C3055 ¥3200E
9784881357019
1923055032002
株式会社 翔泳社
定価：本体 3,200円+税

SE
SHOEISHA

改訂版
独習C++
ハーバート・シルト 著
CD-ROM付
SE
SHOEISHA

# 独習C++
改訂版

SE
SHOEISHA

CD-ROMの内容、使い方に関しては
本文の「付属CD-ROMの使い方」を
必ずお読みください



COMPACT disc